NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

DEVELOPMENT OF A NEW PREDICTION ALGORITHM AND A SIMULATOR FOR THE PREDICTIVE READ CACHE (PRC)

by

F. Nadir Altmisdort

September 1996

Thesis Advisor: Douglas J. Fouts

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1996	Master's Thesis

4. TITLE AND SUBTITLE DEVELOPMENT OF A NEW PREDICTION ALGORITHM AND A SIMULATOR FOR THE PREDICTIVE READ CACHE (PRC)	5. FUNDING NUMBERS
6. AUTHOR(S) F. Nadir Altmisdort	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Efforts to bridge the cycle-time gap between high-end microprocessors and low-speed main memories have led to a hierarchical approach in memory subsystem design. The predictive read cache (PRC) has been developed as an alternative way to overcome the speed discrepancy without incurring the hardware cost of a second-level cache. Although the PRC can provide an improvement over a memory hierarchy using only a first-level cache, previous studies have shown that its performance is degraded due to the poor locality of reference caused by program branches, subroutine calls, and context switches.

This thesis develops a new prediction algorithm that allows the PRC to track the miss patterns of the first-level cache, even with programs exhibiting poor locality. It presents PRC design alternatives and hardware cost estimates for the implementation of the new algorithm. The architectural support needed from the underlying microprocessor is also discussed.

The second part of the thesis involves the development of a memory hierarchy simulator and an address-trace conversion program to perform trace-driven simulations of the PRC. Using address traces captured from a SPARC-based computer system, the simulations show that the new prediction algorithm provides a significant improvement in the PRC performance. This makes the PRC ideal for embedded systems in space-based, weapons-based and portable/mobile computing applications.

14. SUBJECT TERMS Cache, Predictive Read Cache, PRC, Memory, Address Traces, Simulator	15. NUMBER OF PAGES 146
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

DEVELOPMENT OF A NEW PREDICTION ALGORITHM AND A SIMULATOR FOR THE PREDICTIVE READ CACHE (PRC)

F. Nadir Altmisdort
Ltjg, Turkish Navy
B.S., Turkish Naval Academy, 1990

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1996

ABSTRACT

Efforts to bridge the cycle-time gap between high-end microprocessors and low-speed main memories have led to a hierarchical approach in memory subsystem design. The predictive read cache (PRC) has been developed as an alternative way to overcome the speed discrepancy without incurring the hardware cost of a second-level cache. Although the PRC can provide an improvement over a memory hierarchy using only a first-level cache, previous studies have shown that its performance is degraded due to the poor locality of reference caused by program branches, subroutine calls, and context switches.

This thesis develops a new prediction algorithm that allows the PRC to track the miss patterns of the first-level cache, even with programs exhibiting poor locality. It presents PRC design alternatives and hardware cost estimates for the implementation of the new algorithm. The architectural support needed from the underlying microprocessor is also discussed.

The second part of the thesis involves the development of a memory hierarchy simulator and an address-trace conversion program to perform trace-driven simulations of the PRC. Using address traces captured from a SPARC-based computer system, the simulations show that the new prediction algorithm provides a significant improvement in the PRC performance. This makes the PRC ideal for embedded systems in space-based, weapons-based and portable/mobile computing applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. MEMORY HIERARCHY

Improvements in computer architecture and very large scale integrated (VLSI) circuit technology have led to high-end microprocessors with increasing performance and logical complexity. However, the improvement in memory access times has not been sufficient to keep pace with today's high-performance microprocessors. The increasing demand for memory bandwidth has turned the main memory into a major bottleneck for overall system performance.

In order to exploit high-speed microprocessors, designers have taken a hierarchical approach in implementing memory systems. The memory hierarchy consists of multiple levels of memory with different sizes and access times. Each level in the hierarchy contains a subset of the data from the next level in order to keep more recently accessed data items closer to the central processing unit (CPU). The register set of the CPU is considered as the first and the fastest level in the memory hierarchy. The intermediate memory levels between the CPU registers and the main memory are referred to as cache memories.

Even though the concept of a memory hierarchy originated in the early 1960s, the IBM 360/85 was the first commercial computer system to implement a cache memory in 1968 [Ref. 1, p. 486]. The evolution of cache memories continued with minicomputer systems in the 1970s. With the advent of VLSI technology, cache memories were integrated on the same chip as the microprocessor.

Today, most of the high-end microprocessors use an external second-level cache in addition to their first-level on-chip caches. Some more aggressive implementations integrate even the second-level cache inside the same package as the microprocessor. This kind of design provides a wider data path between caches and reduces the number of off-chip transactions. Unfortunately, these implementations are very expensive and usually require a third level off-chip cache. Regardless of how they are implemented, cache memories are

likely to remain as the most cost-effective solution to the memory latency problem in the near future.

B. CACHE THEORY

The cache theory takes advantage of a program behavior known as the locality of reference. Hennessy and Patterson define the principle of locality in two dimensions: temporal locality and spatial locality [Ref. 1, p. 403]. According to temporal locality, programs tend to reuse instructions and data that they have used recently. This type of behavior can be expected, especially from program loops. Spatial locality refers to the tendency of programs to reference instructions and data from contiguous memory locations. This behavior is more obvious in instruction references because instructions are mostly executed sequentially.

A cache creates the illusion of a faster main memory by providing most of the instructions and data requested by the CPU. The CPU can continue to execute at the speed of the cache provided that the cache contains a valid copy of the requested data. This is called a cache hit. However, if the data is not present in the cache, a miss occurs and a main memory access starts for the CPU request. The frequency with which the requests hit in the cache is called the hit ratio. The miss ratio, on the other hand, is calculated by simply subtracting the hit ratio from the unity [Ref. 1, p. 404].

The hit ratio has been the most widely used measure of cache performance in the early cache studies. However, an evaluation based solely upon the hit ratio can be misleading because the cache exhibits the same hit ratio regardless of the speed of the main memory or of the CPU. The overall system performance depends not only on the hit ratio but also on the miss penalty, which is the time required to update the cache from the main memory. The miss penalty is determined by the memory latency and the bandwidth between the cache and the main memory. Contemporary cache studies use the average memory access time in order to observe the combined impact of the hit ratio, cache access time, and miss penalty on CPU performance [Ref. 1, p. 405].

The organization of the cache is determined by its size, block size, and associativity. Since a cache is smaller than the main memory, a method is required to translate a memory address into a cache address. Most caches use some low-order bits of the memory address as an index into the cache and store the remaining high-order bits as a tag. The use of tags is necessary to distinguish between different memory locations that map into the same cache location.

The minimum unit of data associated with an address tag in the cache is called a block [Ref. 2, p. 10]. The choice of block size is a critical design decision because data transfers between the cache and the main memory are performed in units of cache blocks. A large block size enables the cache to take advantage of spatial locality by fetching multiple words from contiguous memory locations. However, increasing the block size also means that more words must be fetched from the memory each time a miss occurs in the cache. As a result, the miss penalty increases due to the increase in memory transfer time. A. J. Smith provides a thorough analysis of the choice of the block size for a cache and the impact of this choice on the CPU cycle count in [Ref. 3, p. 1063].

Associativity of the cache specifies the method with which a memory location is mapped into a cache block. The cache is said to be direct-mapped if a memory address is mapped into exactly one location in the cache. A direct-mapped cache is the simplest cache organization with an associativity of one [Ref. 4, p. 502]. A fully-associative cache can map a memory address into any of its blocks at the expense of more complicated hardware and timing requirements. It uses a content addressable memory (CAM) to perform simultaneous tag comparisons in all blocks [Ref. 5, p. 14]. A set-associative cache, on the other hand, is a compromise between the direct-mapped and the fully-associative cache implementations. Each set contains a limited number of blocks into which a memory address can be mapped. The cache is said to be N-way set associative if there are N blocks in each set [Ref. 5, p. 52].

A significant improvement in the hit ratio can be realized through the implementation of higher degrees of associativity. Although increasing the associativity does not affect the cache size, it reduces the thrashing in the cache caused by memory

addresses that map into the same cache block. However, the improvement realized by increased associativity becomes less significant as the cache size increases [Ref. 5, p. 53].

In addition to the organizational parameters, there are operational cache policies such as the replacement policy, the write policy, and the write-miss policy. Cache policies must be chosen carefully, depending on the amount of resources available to improve the design. The replacement policy selects a block to be replaced when a read miss occurs in an associative cache. Direct-mapped caches do not require a replacement policy because there is only one candidate for replacement. The most common replacement policies are *least-recently-used* (LRU) and *random* [Ref. 4, p. 510].

There are two fundamental write policies which define the behavior of the cache during a write cycle. If the data is written to both the block in the cache and the block in the downstream memory (main memory or a second-level cache), the policy is called *write-through*. If the data is written only to the cache without being propagated to the downstream memory hierarchy, the policy is called *write-back* [Ref. 4, p. 511]. With the write-back policy, the writes can be performed at the cache speed. The memory bandwidth is more effectively used because multiple writes into a cache block require only one write to the downstream memory, only when the block is being replaced. However, the implementation of the write-back policy is more costly in hardware than that of the write-through policy [Ref. 5, p. 63].

The cache design is also influenced by the underlying microprocessor architecture. In computer systems using virtual addressing [Ref. 4, p. 481], caches can be located either upstream or downstream of the microprocessor's memory management unit (MMU). If the cache is located upstream of the MMU, it is called a *virtual* or *logical* cache. If the cache is located downstream of the MMU, it is called a *physical* cache [Ref. 5, p. 49]. Virtual caches require the handling of a problem known as aliasing, caused by the mapping of more than one virtual address into the same physical address [Ref. 4, p. 493].

Any particular level in the memory hierarchy can accommodate either a unified cache or two split caches for instruction and data references. A split cache organization

enables independent optimization of the instruction and data caches [Ref. 5, p. 60]. Most modern computers use split instruction and data caches in the first level followed by a unified cache in the second-level.

Researchers have been studying different aspects of cache theory for more than 30 years by using either analytical models or simulations. However, the large number of parameters involved in optimizing cache memories and the recent improvements in computer architecture leave much room for future research.

C. THE PREDICTIVE READ CACHE

The Predictive Read Cache (PRC) is a special-purpose cache memory that is logically inserted between the first-level data cache and the main memory. It predicts the address of the next read miss in the data cache by using a displacement-based prediction algorithm. The design, operation, and implementation of the PRC is described by Fouts and Billingsley along with a number of performance results obtained from simulations of the PRC [Ref. 6].

The idea of the PRC is based upon the concept of a memory prediction buffer (MPB), which is placed between the main memory and the data cache to reduce the main memory latency [Ref. 7]. After examining several data-cache read-miss patterns, Fouts and Billingsley concluded that the MPB and its simple prediction algorithm is unable to follow the temporal interleaving of the various different address traces, although it is able to follow the spatial locality of each individual address trace [Ref. 6, p. 112]. The PRC has been developed in an effort to overcome the shortcomings of the MPB with its ability to track multiple read miss patterns in the first-level data cache.

The prediction algorithm first calculates a signed displacement between the addresses of two consecutive read misses in the first-level data cache. This displacement is then added to the most recent read miss address to obtain the predicted address of the next read miss in the first-level data cache [Ref. 6, p. 110]. The PRC makes a new prediction each time another read miss occurs in the first-level cache. If, however, a cache miss also misses in the PRC, then a new pattern is started in a different block.

The PRC can use a direct-mapped, set-associative, or fully-associative mapping like any other cache memory. In addition to the standard address tags and data memory, the PRC is provided with additional storage to accommodate the most recent miss address and the previous miss address for each block. The hardware required to implement the prediction algorithm consists of a single subtracter-adder pair, owing to the fact that only one prediction is needed in a particular PRC block at any given time [Ref. 6, p. 117].

The simulations performed by Fouts and Billingsley show that the performance of a fully-associative PRC using a random replacement policy is either better than or close to that of a second-level cache [Ref. 6, p. 117].

A series of trace-driven simulations has recently been performed by Miller in order to extend previous efforts to measure the PRC performance [Ref. 8]. Miller has used address traces collected from an Intel 486 processor at Brigham Young University (BYU) [Ref. 9, p. 450]. After determining the baseline performance of a system with only a first-level cache, Miller has simulated the effects of adding a PRC to the memory hierarchy. His results are consistent with those obtained by Fouts and Billingsley, revealing that the PRC can outperform a second-level cache of comparable size due to its predictive nature [Ref. 8, p. 37].

D. OUTLINE OF THESIS

Although the PRC can provide a significant improvement to the overall system performance, previous studies have shown that its performance is degraded with poor locality of reference caused by branches, subroutine calls, and context switches under multitasking workloads [Ref. 6, 8]. This thesis develops a new prediction algorithm in an effort to improve the PRC performance by making it less sensitive to programs exhibiting poor locality. It also develops a memory hierarchy simulator and an address-trace conversion program to perform trace-driven simulations of the PRC.

Chapter II introduces the fundamentals of the new prediction algorithm and the architectural support needed from the underlying microprocessor. It discusses the PRC

design alternatives employing direct-mapped, set-associative, and fully-associative organizations.

Chapter III presents the hardware cost estimates, based on the number of transistors associated with the design alternatives introduced in Chapter II.

Chapter IV discusses the development of software tools that are used to establish a simulation environment for the new prediction algorithm. These tools include the BACH Address Trace Editor (BATE) and the Trace Converter (Tracer) which are used to perform address trace conversions.

Chapter V introduces the design, software architecture, and operation of a simulation program, namely the Cache and PRC Simulator (CaPSim), that is used for the trace-driven simulations.

Chapter VI presents the simulation test cases and documents the results obtained from these simulations.

Finally, Chapter VII is the conclusion of the thesis.

II. DESIGN OF A NEW PREDICTIVE READ CACHE

A. A NEW PREDICTION ALGORITHM

The development of a new algorithm has been proposed by Professor Douglas J. Fouts of the Naval Postgraduate School in order to improve the PRC performance under multitasking workloads. Even though the PRC outperforms the MPB by tracking multiple read miss patterns, its performance is still sensitive to the effects of program branches and context switches, which cause systematic pollution in the PRC [Ref. 6, 8]. Miller has already shown that changing the PRC size, associativity, or other parameters such as write policy and write miss policy does not provide a significant increase in performance [Ref. 8, pp. 24-34]. Therefore, the PRC performance can be improved only if a new algorithm can be implemented so that the PRC can continue to retain multiple address traces without being affected from irregular memory access patterns. On the other hand, any modification of the prediction algorithm must result in reasonable hardware complexity, since low implementation cost of the PRC is its major advantage over a second-level cache.

The idea behind the new algorithm is to maintain a relationship in the PRC between the addresses of read misses and the addresses of instructions that cause these read misses. In this design, each PRC block is tagged with an instruction address in order to track read miss trends of different instructions. For RISC architectures, load instructions are of primary interest. Figure 1 shows the logical contents of a single PRC block using the new prediction algorithm. All fields except the *instruction address tag* are inherited from the original PRC design [Ref. 6, p. 119].

The prediction in each PRC block is performed by using the simple displacement algorithm of the MPB [Ref. 6, p. 110]. A signed displacement is calculated between the most recent miss address (MRMA) and the previous miss address (PRMA). This displacement is then added to the MRMA to obtain the predicted address of the next read miss in the first level data cache.

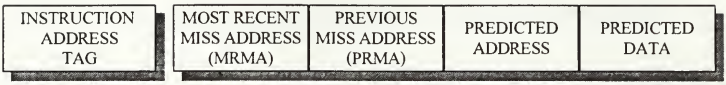

Figure 1. Logical Contents of a PRC Block

Figure 2 shows the location of the PRC in the memory hierarchy. The only difference from the previous design is the *instruction address bus* inserted between the CPU and the PRC. The conventional address bus is denoted as the *data address bus* to distinguish between two separate address busses. Since the PRC is only used for data references, the instruction cache is not included in the figure.

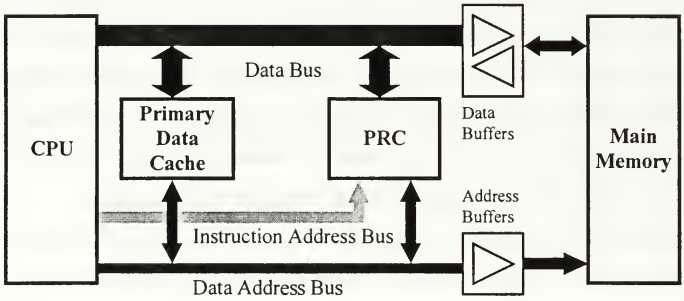

Figure 2. PRC Location in Memory Hierarchy

The modified prediction algorithm does not affect the functional description of the PRC from the perspective of the cache or main memories [Ref. 6, p. 113]. At the beginning of a read cycle, the CPU places the memory read address on the *data address bus* and the corresponding instruction address on the *instruction address bus*. The dedicated bus between the CPU and the PRC is completely transparent to the primary data cache. If the read request hits in the data cache, no action is taken by the PRC. However, if a read miss occurs in the data cache, the PRC starts a read access and sends the read request to the main memory in case it misses in the PRC as well. The PRC simultaneously compares the

instruction address tag and the predicted address field against the addresses placed on the *instruction address bus* and the *data address bus*, respectively. A match is required in both of these fields to qualify the request as a PRC read hit. Assuming that the data cache and the PRC both use the same block size, the PRC can update the data cache in a single cycle when a read hit occurs. The required data is also forwarded to the CPU while the initiated main memory read cycle is canceled. The PRC then calculates the next predicted address where the current instruction is expected to miss again and starts a prefetch cycle from the main memory. If the instruction address misses in all PRC blocks, a new trace is started for the instruction in a selected block. The PRC cannot make a prediction until the instruction misses once more in the data cache. Since the predictions are associated with instructions, most of the address traces will be preserved in the PRC after a context switch or subroutine call.

An important requirement in designing a memory hierarchy is to keep the intermediate memory levels consistent with the main memory. This argument places an emphasis on handling memory writes in cache memories. The use of a PRC within the memory hierarchy does not place any limitations on the type of the write policy that can be used with the primary data cache [Ref. 6, p. 113]. However, Fouts and Billingsley indicate that the PRC itself cannot use a write-back policy because the main memory might not get updated for a considerable period of time [Ref. 6, p. 114]. The PRC can use a write-through policy so that the data is written into the corresponding PRC block on a write hit and is ignored on a write miss. It can also use a write-update or a write-invalidate policy which are special cases of the write-through policy [Ref. 5, p. 62]. These policies imply that the PRC must be either updated (write-update) or invalidated (write-invalidate), regardless of the write cycle being a hit or a miss in the PRC.

There are two basic write miss policies that can be used in conjunction with a write policy: *write-around* and *write-allocate* [Ref. 1, p. 413]. With a write-allocate policy, if the size of the data being written is smaller than the block size, the missing portion of the block is fetched from the main memory before the data is written into the block. The write-

allocate policy is not beneficial with a write-through policy because the subsequent writes to the allocated block must still be propagated to the upper memory levels [Ref. 1, p. 414]. Therefore, the PRC uses a write-around policy which simply bypasses the writes on a write miss.

B. ARCHITECTURAL SUPPORT

The new PRC design requires some support from the microprocessor architecture. The PRC must be provided with the address of the instruction that causes a memory read, in addition to the address of the read itself. Since the PRC is designed as an on-chip component, the external interface of the microprocessor will not be affected by this requirement. With some minor modifications to the CPU architecture, a dedicated internal bus can be used for instruction addresses.

Figure 3 depicts the block diagram of a simple RISC microprocessor using on-chip instruction and data caches and an on-chip predictive read cache. The microprocessor has a decode/dispatch unit and three execution units operating on a register file. The bus interface provides the external access for all three cache memories.

The instructions are fetched from the instruction cache by the decode/dispatch unit through a prefetch queue. The program counter (PC) contains the address of the instruction being decoded in any given cycle. If the instruction is a memory load, the decode/dispatch unit simply stores the contents of the program counter into the instruction address register (IAR) and dispatches the instruction to the load/store unit for the effective address calculation. As soon as the load/store unit places the effective address into the memory address register (MAR), a memory read cycle is initiated by simultaneously enabling the contents of the IAR and MAR on the corresponding busses. The read cycle is terminated when the requested data is latched into the memory data register (MDR). It should be noted that the instruction address need not be sent to the PRC during a memory write cycle because the PRC does not make any predictions for memory writes.

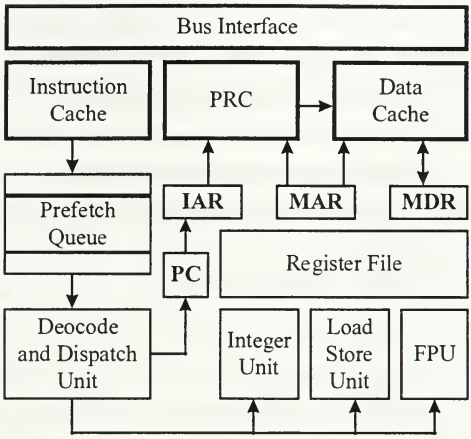

Figure 3. Architectural Support for the New PRC Design

The previous discussion indicates that the hardware support needed for the new PRC design can be provided by using an additional CPU register (IAR) and a dedicated address path connecting the output of the register to the PRC. However, the use of a superscalar microprocessor with out-of-order execution would require more complicated hardware measures for integrating the new PRC with the CPU.

C. DESIGN ALTERNATIVES

Although the mapping algorithm of the new PRC operates on instruction addresses, the PRC can still be designed as a direct-mapped, set-associative, or fully-associative cache. Before starting a PRC design, at least three of the four organizational parameters must be

specified [Ref. 2, p. 22]. These parameters are the PRC size, the block size, the degree of associativity, and the number of sets in the PRC. The PRC size (or the cache size in general) is interpreted as the size of the data memory alone, without including the size of the tag memory or the additional storage used for bookkeeping purposes [Ref. 5: p. 12].

Figure 4 shows the organization of a typical data memory with a block size of B bytes and an associativity of a. The associativity specifies the number of blocks in each set. Given a PRC size of P, the number of sets can be calculated by using Equation 1

$$\text{Number of Sets} = s = \frac{P}{aB} \tag{1}$$

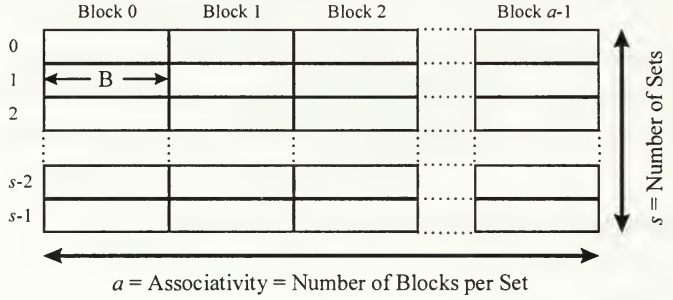

Figure 4. PRC Data Memory

The number of sets determines the mapping function of the PRC in the direct-mapped and set-associative designs. Figure 5 illustrates how an instruction address is partitioned into two fields depending on the mapping function. Assuming byte addressing with addresses that are 32 bits long and aligned on word boundaries, the least significant two bits of the instruction address should always be zero. As a result, only the most significant 30 bits of the instruction address are decoded by the PRC.

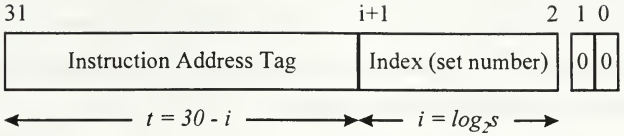

Figure 5. Instruction Address Mapping

Unlike conventional cache memories and the original PRC, the new PRC does not use the block size in its mapping function. Normally, the block size determines the least significant address field known as the block offset [Ref. 1, p. 411]. If there are 16 bytes in a block (B), then the size of the block offset is four bits (log_2B). However, the bytes within a PRC block are selected by using the data address, not the instruction address. Therefore, the block offset field appears in the data address as shown in Figure 6. The rest of the data address is used as the data address tag, since the index field is already part of the instruction address. This implies that the new PRC must perform two separate tag comparisons in parallel.

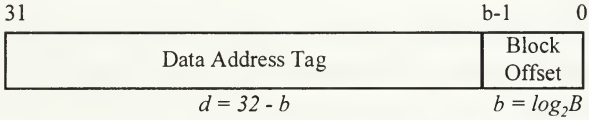

Figure 6. Data Address Mapping

In the following subsections, each design alternative is explained at the register-transfer level (RTL) by using specific organizational parameters to clarify the design process.

1. Direct-Mapped PRC Design

The logical organization of a direct-mapped PRC is shown in Figure 7. The first two fields are the instruction address and the data address tags. The data address tag is, in fact, the predicted address of the next read miss in the primary data cache. The MRMA field is the same size as the predicted address and contains the address of the most recent read miss.

The new PRC stores a single miss address for prediction and eliminates the previous miss address (PRMA) field used in the original PRC design [Ref. 6]. This elimination reduces the hardware cost without any degradation in the performance. The operational details will be explained with an example later in this section.

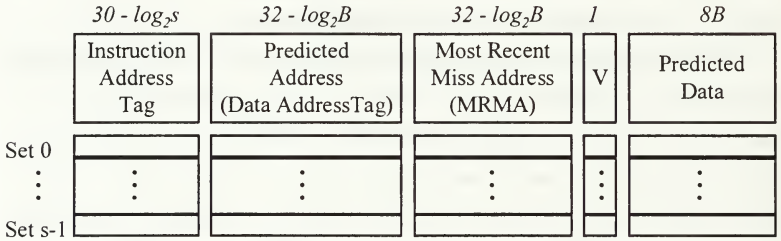

Figure 7. Logical Organization of a Direct-Mapped PRC

The predicted data field represents the data memory of the PRC. Each block contains *B* consecutive bytes prefetched from the predicted address in the main memory. The valid bits (V) indicate whether the data stored in a block is consistent with the contents of the main memory.

An instruction address can be mapped into a single block in a direct-mapped PRC. Selecting the boundary between the tag and the index fields of the instruction address is an important design decision, involving the optimization of organizational parameters. One of the extremes in this selection would be using a single block in the PRC, with all 30 bits of the instruction address being used as a tag. This approach will result in intolerable thrashing in the PRC because all instructions that miss in the first-level data cache will replace each other. The other extreme would be using all 30 bits of the instruction address as an index, resulting in 2^{30} (*1,073,741,824*) blocks. In this case, each instruction can be mapped into an individual block without any conflicts. However, the hardware cost of this gigantic PRC is prohibitive. Therefore, a reasonable boundary must be selected between the tag and the index fields in order to obtain the best cost/performance trade-off in the PRC design.

The operation of the PRC can be explained by using an example design. Figure 8 shows the RTL layout of a 2-Kbyte PRC with a block size of 16 bytes and an associativity of one. The PRC contains 128 sets, calculated by substituting the PRC parameters into Equation 1. The total number of blocks is equal to the number of sets, since there is only one block in each set.

The PRC interfaces with three busses: DA(31:0) is the 32-bit *Data Address Bus*, IA(31:2) is the 30-bit *Instruction Address Bus*, and DT(127:0) is the 128-bit *Data Bus*. There are four separate sections of the PRC indexed with bits IA(8:2). The *Instruction Tag* memory contains the 23-bit instruction address tags, the *Predicted Address* memory contains the 28-bit predicted address tags, the *MRMA* memory contains the 28-bit most recent miss addresses, and the *Predicted Data* memory contains the 16-byte data prefetched from the main memory.

The predictor is a cascaded 28-bit subtracter/adder pair, which calculates the next read miss address by first finding a displacement between the last two miss addresses and then adding this displacement to the most recent miss address. The 28-bit *Data Address Register* and the 28-bit *Predicted Address Register* are connected to the input and output ports of the predictor, respectively. There are also two comparators to determine whether the instruction address tag and the corresponding data address tag match in the PRC. The 23-bit comparator is used for the instruction address tags and the 28-bit comparator is used for the data address tags. The PRC controller, the valid bits, and the address decoder are not shown in Figure 8 for clarity.

The behavior of the PRC depends on the output signals *I-hit* and *D-hit*, which are generated by two comparators. Table 1 summarizes the possible operating modes of the PRC for different values of these two signals after a PRC read access is completed.

In the *Total PRC Miss* case, neither the instruction address tag IA(31:9) nor the data address tag DA(31:4) matches the values contained in the PRC. A block replacement must be performed by loading the instruction address tag IA(31:9) into the selected block in the

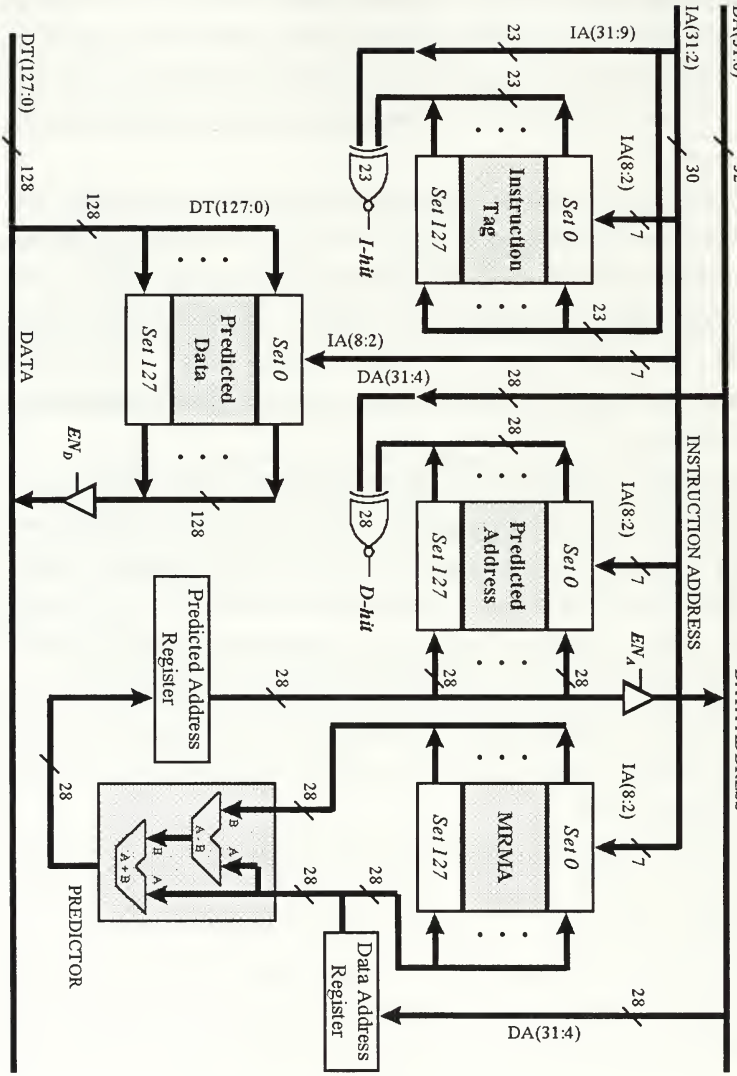

Figure 8. Direct-Mapped PRC Design

instruction tag memory. However, no replacement policy is required because the PRC has a direct-mapped organization. The most significant 28 bits of the data address, DA(31:4), must also be loaded into the MRMA memory as the most recent miss address. For the first cache read miss, the PRC cannot make a prediction because of the fact that at least two read misses are required in the same block before a displacement can be calculated. No data can be forwarded to the primary data cache in the *Total PRC Miss* mode. Therefore, the cache must be updated from the main memory.

I-hit	D-hit	Description	Line Replacement	Prediction	Forward Predicted Data
0	0	Total PRC Miss	Yes	No	No
0	1	Partial PRC Hit	Yes	No	Yes (if valid)
1	0	Partial PRC Miss	No	Yes	No
1	1	Total PRC Hit	No	Yes	Yes (if valid)

Table 1. PRC Operating Modes

The *Partial PRC Hit* case is unique to the new PRC design. Even though the instruction address tag misses in the PRC, a hit occurs in the predicted address memory. This situation may result from a load instruction missing at an address which is exactly the same as the predicted address for a previous load instruction. Of course, both instruction addresses must have an identical index field, IA(8:2), because they map into the same block. Although this is an extraordinary case, the data can still be forwarded to the primary cache, provided that it is valid. However, a prediction cannot be made since this is normally the first read miss for the current instruction and the value of the *MRMA* memory is the most recent miss address of the previous instruction. A block replacement must take place, as explained in the *Total PRC Miss* case.

As far as the primary data cache and the CPU are concerned, the *Partial PRC Miss* case is the same as the *Total PRC Miss* because no predicted data can be forwarded by the PRC. However, the PRC treats this case in a different manner because the instruction tag hits in the PRC. In general, a *Partial PRC Miss* may occur under two conditions:

- When a load instruction misses for the second time in the PRC, a *Partial PRC Miss* occurs. In other words, for a particular load instruction, a *Total PRC Miss*

is always followed by a *Partial PRC Miss*, unless the instruction address is overridden during a block replacement before it misses for the second time.

- A *Partial PRC Miss* may occur due to the misprediction of the next read miss address for a particular load instruction.

A line replacement is not required in this case because the instruction is already in the PRC. However, a prediction must be made. Every time a request is received from the CPU, the data address bits DA(31:4) are latched into the data address register. This is, essentially, the most recent miss address, while the corresponding value in the MRMA memory becomes the previous miss address. When a prediction cycle is started, the predictor calculates a displacement by subtracting the previous miss address from the most recent miss address. Then, it adds the value of the displacement (either positive or negative) to the most recent miss address to calculate the next read miss address in the primary data cache. The predictor operates on the most significant 28 bits of the addresses because the block offset is 4-bits long. The address calculated by the predictor is temporarily stored in the predicted address register. As soon as the main memory read cycle that has been initiated by the primary data cache is completed, the PRC starts a prefetch cycle by putting the content of the predicted address register on the data address bus. The least significant four bits of the bus must be driven to zero by the PRC. The predicted address is also stored into the corresponding block in the predicted address memory. When the prefetch cycle is completed, the data returned by the main memory is stored in the data memory and the corresponding valid bit is set.

When both the instruction address tag and the data address tag hit in the PRC, a *Total PRC Hit* occurs. The predicted data is forwarded to the primary data cache and the CPU if the corresponding valid bit is set. Address prediction is performed as explained above.

2. Set-Associative PRC Design

The set-associative PRC design is quite similar to the direct-mapped design, except that there is more than one block in each set. Figure 9 shows the RTL layout of a 4-way,

set-associative PRC with the same parameters as the direct-mapped design. Even though the total number of blocks remains unchanged, the number of sets is reduced by a factor of four. Each SRAM contains 32 sets and each set contains four blocks.

The reduction in the number of sets also affects the mapping function so that each SRAM is indexed with the bits IA(6:2) instead of IA(8:2) of the direct-mapped design. The instruction address tag becomes IA(31:7), while the data address tag remains unchanged as DA(31:4). There are four blocks in each set, therefore, the instruction tag memory and the predicted address memory have four comparators that operate in parallel. The results of the instruction tag comparisons (*I-hit*) are used by the *Block Replacement Unit* to generate the block selector bits. The block selectors also enable the outputs of the corresponding data address tag comparators to form the final *D-hit* output. The interpretation of the *I-hit* and *D-hit* combinations is the same as the direct-mapped design.

One major difference from the direct-mapped design is the addition of the *Status Memory,* used to store the block replacement status. The replacement policy in this example is chosen as the pseudo-LRU algorithm, which was developed by the Intel Corporation for the i486 microprocessor [Ref. 5, p. 57]. The pseudo-LRU is a cost-effective alternative to the true LRU algorithm [Ref. 1, p. 411] with a comparable improvement in the performance. As the degree of associativity is increased, the LRU replacement can consume a considerable amount of hardware. For an *m*-way set-associative cache, $\lceil \log_2(m!) \rceil$ status bits are required to encode all of the access patterns in a set [Ref. 5, p. 57]. Another problem encountered in the LRU scheme is the need for a read-modify-write cycle on the status bits for each cache access in order to maintain the order of the most recent accesses [Ref. 5, p. 58]. This may impose some timing restrictions on the PRC implementation. On the other hand, the pseudo-LRU algorithm requires only *m-1* status bits for an *m-way* set-associative cache. Therefore, three status bits are needed in the 4-way, set-associative design, denoted as P_2, P_1, and P_0 in Figure 9.

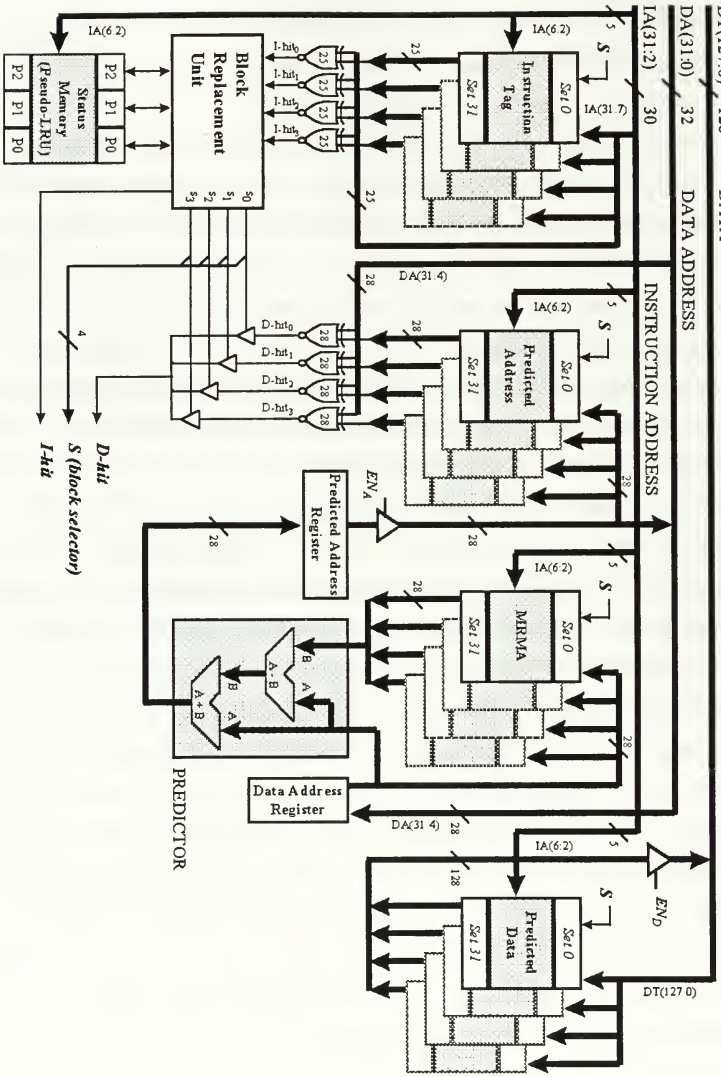

Figure 9. Set Associative PRC Design

In general, the status information is stored as a binary tree that consists of $\lceil \log_2 m \rceil$ levels. Figure 10 shows the binary tree structure for the 4-way, set-associative PRC. In the first level, if a hit occurs in either way 3 or way 2, the status bit P_2 is set. It is cleared upon a hit in way 1 or way 0. In the second level, P_1 is set if a hit occurs in way 3 and cleared if a hit occurs in way 2. However, the value of P_0 remains unchanged. Similarly, P_0 is set upon a hit in way 1 and cleared upon a hit in way 0, without changing the value of P_1. The truth table associated with this binary tree is given in Table 2.

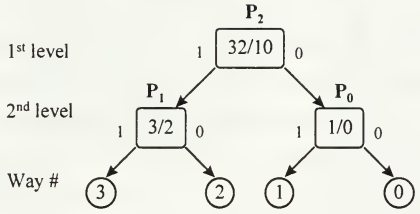

Figure 10. Pseudo-LRU Replacement

I-hit$_3$	I-hit$_2$	I-hit$_1$	I-hit$_0$	P_2	P_1	P_0
1	0	0	0	1	1	*
0	1	0	0	1	0	*
0	0	1	0	0	*	1
0	0	0	1	0	*	0

Table 2. Status Update Logic

When IA(31:7) misses in all ways, the *Block Replacement Unit* selects a victim block based upon the latest status information stored in the *Status Memory* for a particular set. The truth table for victim selection is given in Table 3.

P_2	P_1	P_0	s_3	s_2	s_1	s_0
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	1	0	0	1	0	0
0	1	1	0	1	0	0
1	0	0	0	0	1	0
1	0	1	0	0	0	1
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Table 3. Victim Selection Logic

Figure 11 depicts a simplified logic-level diagram for the *Block Replacement Unit*. A single status memory set is also shown in order to demonstrate the interface between the *Status Memory* and the *Block Replacement Unit*. It must be noted that the *Status Memory* is not associative. Each of its sets contains the 3-bit status information for all ways in the corresponding PRC set. The status bits can be read or written individually by using two enable signals, provided that the corresponding set is also enabled.

The *Block Replacement Unit* samples the outputs of the instruction tag comparators (*I-hit*$_0$, *I-hit*$_1$, *I-hit*$_2$, and *I-hit*$_3$) at the positive edge of the *select* input, which is assumed to be asserted by the PRC controller. These bits are logically ORed to obtain the value of the *I-hit* output. If a read hit occurs in a set, the status bits are updated by the status update logic and the block selector outputs are determined by the values of corresponding *I-hit* bits. Otherwise, the victim selection logic selects a block to be replaced. During an update, the write enables (*ENwrite*) of the status bits P_1 and P_0 are conditionally asserted. However, all status bits are read with a single enable signal (*ENread*) during victim selection. Since the pseudo-LRU algorithm does not require a read-modify-write cycle on status bits, they are either read or written, depending on the value of *I-hit*.

Once the values of *I-hit* and *D-hit* are determined, the actions taken by the PRC are the same as those listed in Table 1 for the direct-mapped PRC design.

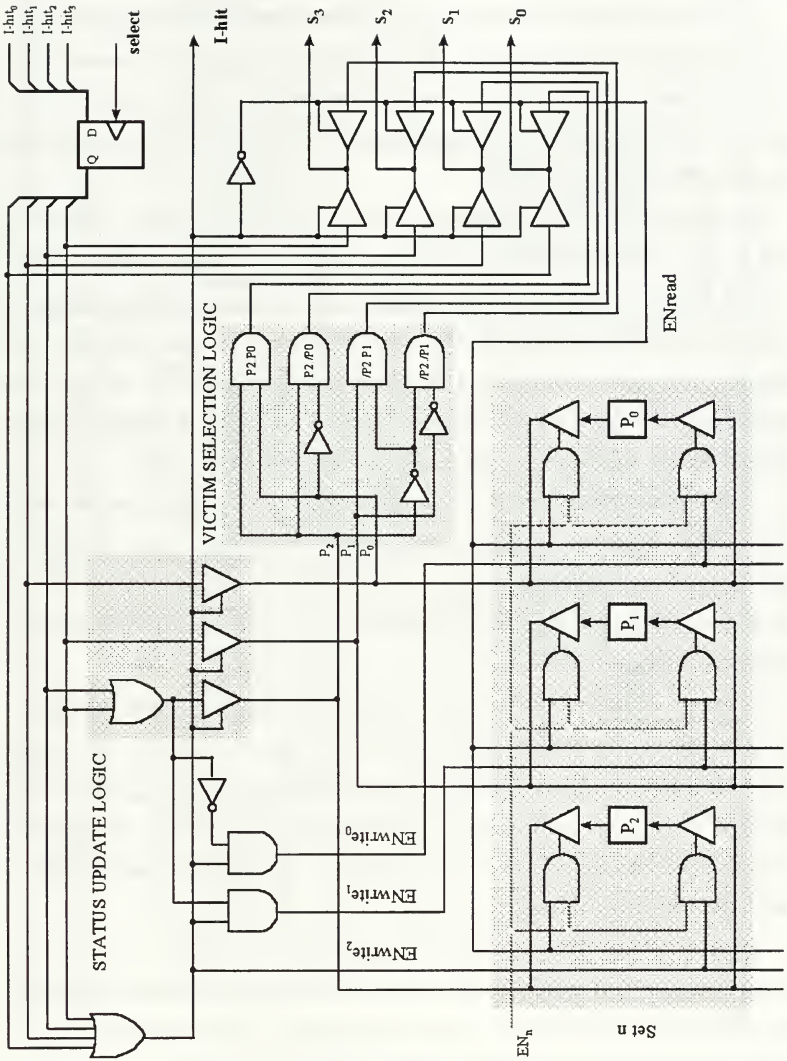

Figure 11. Block Replacement Unit

3. Fully-Associative PRC Design

The fully-associative PRC design is quite different from the direct-mapped and the set-associative designs. There is only one set in the PRC and the degree of associativity is equal to the total number of blocks. Therefore, the term *line* is more appropriate for referring to the blocks of a fully-associative PRC. The number of lines is determined by dividing the PRC size by the block size.

Figure 12 shows the RTL layout of a 4-Kbyte, fully-associative PRC with a block size of 16 bytes. This configuration yields 256 lines (4096÷16) in each section of the PRC. All 30 bits of the instruction address are used as a tag. The instruction tag memory is a content-addressable memory (CAM) in which each line is integrated with a dedicated 30-bit comparator. The value of IA(31:2) is compared with the contents of all 256 lines simultaneously. If a hit occurs in a particular line, the same line in all other memory modules is selected. Otherwise, a victim line is selected to store the new entry.

The *Encoder* generates an 8-bit *Line Address*, LA(7:0), according to the match outputs from the instruction tag memory. *I-hit* gives the overall match output of the PRC. When IA(31:2) misses in all lines, *I-hit* will be low and the line address bus will be driven by the *Line Manager*. The rest of the PRC components function the same way as described in the previous designs.

The fully-associative PRC of Figure 12 acts like a 256-way set-associative PRC. The implementation of a true LRU replacement policy requires $\lceil \log_2(256!) \rceil$ bits to keep track of least recently used lines. Although a pseudo-LRU policy can still be employed for smaller PRC sizes, a different replacement approach will be presented for this particular PRC design by combining the pseudo-LRU replacement with a First-In-First-Out (FIFO) replacement policy [Ref. 1, p. 412].

The logical block diagram of the *Line Manager* is given in Figure 13. The 256-line instruction tag memory is partitioned into four 64-line groups. A FIFO replacement policy is implemented within each group by using four 6-bit binary counters. However, a group is selected as a victim according to the pseudo-LRU policy.

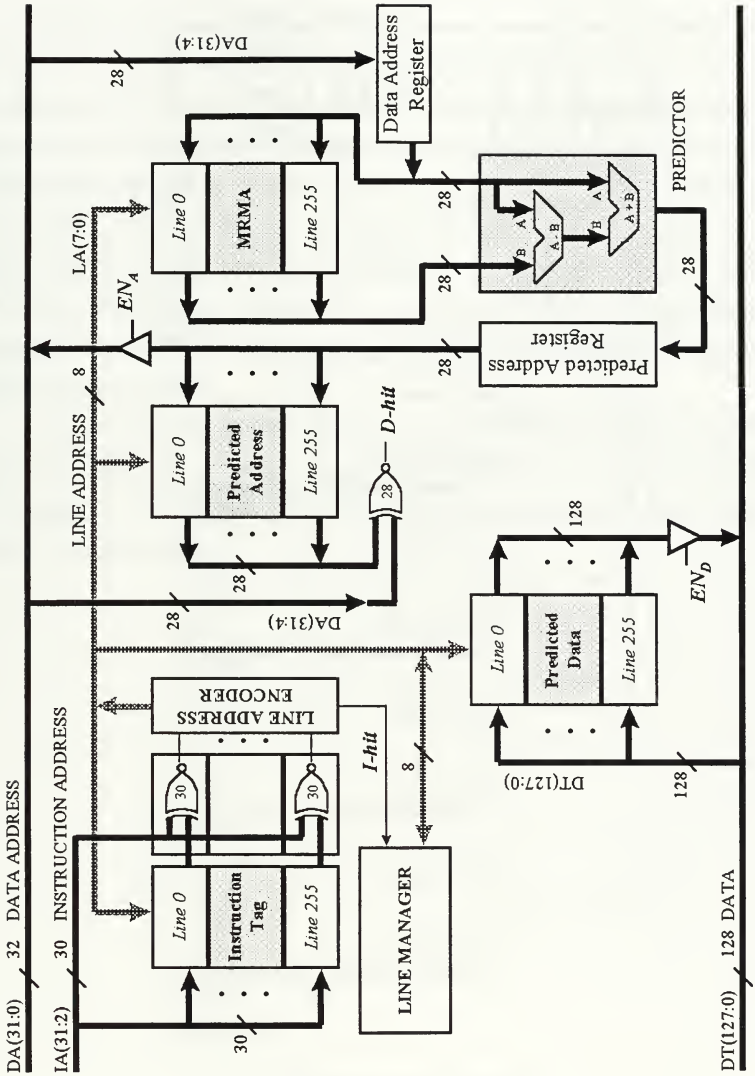

Figure 12. Fully-Associative PRC Design

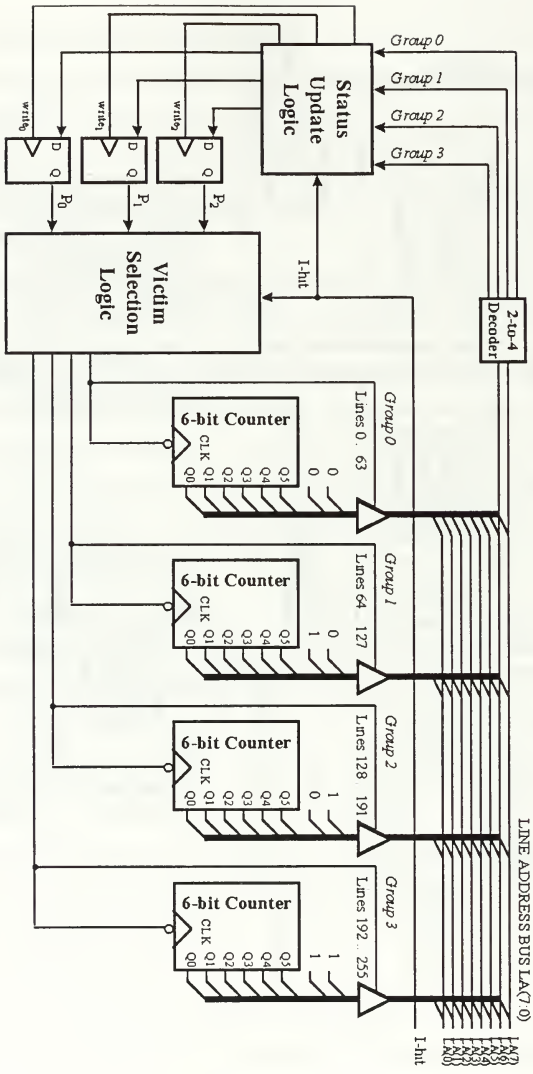

Figure 13. Line Manager

As long as a hit is obtained in the instruction tag memory, the *Line Manager* will only update the status bits (P_2, P_1, and P_0) according to the most significant two bits of the line address, LA(7:6). These two bits are decoded with a 2-to-4 decoder to generate four group inputs to the status update logic. When a miss occurs in the instruction tag memory, a victim group is selected by the victim selection logic, which enables the output of the corresponding counter to drive the line address bus. Each counter is triggered at the negative edge of the group enable signal to advance to the next line in the group. In this way, the line replacement is performed in a first-in-first-out fashion in the least recently used group. One of the four PRC operation modes in Table 1 is assumed after the signals *I-hit* and *D-hit* are determined.

The design alternatives presented in this chapter imply that the migration from the data-address-tagged PRC to the instruction-address-tagged PRC can be accomplished with the addition of a SRAM containing instruction address tags and with some minor changes to the PRC controller logic.

III. HARDWARE COST ESTIMATES

The PRC design requires a trade-off between hardware complexity and the improvement in overall system performance. The VLSI implementation costs of the new design must be reasonable so that the PRC can maintain its advantage over a second-level cache. This chapter presents the hardware cost estimates for the PRC in terms of the number of transistors required for VLSI implementations.

The architecture of a memory chip is shown in Figure 14 to demonstrate the contribution of different components to the total memory cost. The memory layout is determined by the mapping function derived from the organizational parameters of the PRC. A row decoder selects one of the 2^i sets, where i is the number of index bits. The column decoder selects a 4-byte word out of B bytes in each set and the word multiplexer connects the selected word to the data path of the memory. The data is read and written through the sense amplifiers and write buffers, respectively [Ref. 10, p. 565].

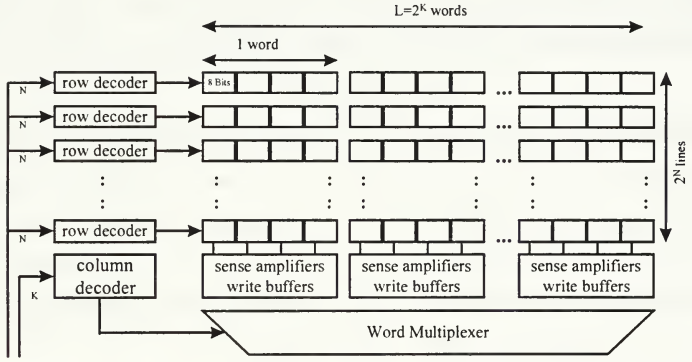

Figure 14. Memory Architecture After [Ref. 10]

The total cost for a particular memory is given by Equation 2 in terms of the number of transistors. The cost of the memory cells, C_{cell}, is the most dominant factor in the total

cost. C_{rd} and C_{cd} represent the costs of row decoders and the column decoder, respectively. The combined cost of sense amplifiers and write buffers is denoted with C_{sw} and the word multiplexer cost is denoted with C_{mux}. The column decoder and the word multiplexer costs must be included only in the data memory cost, since no word selection and multiplexing is needed for the tag memories.

$$C_{memory} = C_{cell} + C_{rd} + C_{cd} + C_{sw} + C_{mux} \quad \textbf{(2)}$$

Assuming a 6-transistor SRAM cell using cross-coupled inverters [Ref. 10, p. 565], Equation 3 gives the cost of memory cells as a function of the number of rows (2^i), the number of bits per row (N), and the cost of a single memory cell (C_{bit}). The number of bits per row is $N=8B$ for the predicted data memory, $N=30-i$ for the instruction tag memory, and $N=32-\log_2 B$ for both the predicted address and the MRMA memory.

$$C_{cell} = 2^i N C_{bit} = 2^i N 6 \quad \textbf{(3)}$$

The cost of row decoders is the same for all memories in the PRC. Assuming that the row decoders are implemented as complementary AND gates [Ref. 10, p. 575], each row requires an i-input AND gate followed by an inverter at the output. Equation 4 gives the total row decoder cost as a function of the number of index bits.

$$C_{rd} = 2^i(2i+2) \quad \textbf{(4)}$$

The cost of sense amplifiers and write buffers depends on the number of bits per row, N, regardless of the number of sets in each memory. A sense amplifier is a 5-transistor differential amplifier that amplifies the voltage difference between the bit lines [Ref. 10, p. 570]. A write buffer, on the other hand, is made of two n-channel pass transistors and two cascaded inverters, resulting in a total of six transistors [Ref. 10, p. 573]. Equation 5 gives the total cost of sense amplifiers and write buffers.

$$C_{sw} = 5N+6N = 11N \quad \textbf{(5)}$$

The column decoder operates on the word bits (w) of the address, rather than the index bits (i) used by a row decoder. Since there are four bytes in a word, the number of word bits can be calculated by simply subtracting two from the size of the block offset. It

should be noted that a column decoder is not necessary if there is a single word in each row. If there are two words per row, then the column decoder can be implemented with an inverter. Otherwise, the column decoder is made of a *w*-input AND gate followed by an inverter [Ref. 10, p. 576]. Equation 6 gives the column decoder cost as a function of the number of word bits.

$$C_{cd} = \begin{cases} 0, & w = 0 \\ 2, & w = 1 \\ 2w+2, & w \geq 2 \end{cases} \quad (6)$$

The word multiplexer can be implemented with pass transistor logic [Ref. 10: p. 304] by using a single transistor for each bit. Therefore, the multiplexer cost is equal to the number of bits per row (*N*).

The transistor cost associated with each SRAM can be calculated as a function of the PRC size (*P*), the block size (*B*), and the degree of associativity (*a*) by substituting $i = log_2(P/aB)$ into Equation 3 and 4, and $w = log_2(B) - 2$ into Equation 6. C_{data} is the predicted data memory cost, C_{itag} is the instruction tag memory cost, C_{dtag} is the predicted address memory cost, and C_{mrma} is the MRMA memory cost. The number of bits in a row (*N*) differs from one memory to the other, as indicated in Equation 7 through 9.

$$\begin{aligned} C_{data} &= a\left(\frac{P}{aB}(6N)+\frac{P}{aB}\left(2\log_2\left(\frac{P}{aB}\right)+2\right)+11N+N+(2w+2)\right), \text{ for } N=8B \\ &= \frac{P}{B}\left[48B+2\log_2\left(\frac{P}{aB}\right)+2\right]+96aB+2a\left(\log_2 B-1\right) \end{aligned} \quad (7)$$

$$\begin{aligned} C_{itag} &= a\left(\frac{P}{aB}(6N)+\frac{P}{aB}\left(2\log_2\left(\frac{P}{aB}\right)+2\right)+11N\right), \text{ for } N=30-\log_2(P/aB) \\ &= \frac{P}{B}\left[182-4\log_2\left(\frac{P}{aB}\right)\right]+330a-11a\left(\log_2\left(\frac{P}{aB}\right)\right) \end{aligned} \quad (8)$$

$$C_{dtag} = C_{mrma} = a\left(\frac{P}{aB}(6N)+\frac{P}{aB}\left(2\log_2\left(\frac{P}{aB}\right)+2\right)+11N\right), \text{ for } N=32-\log_2 B$$
$$= \frac{P}{B}\left[194-6\log_2 B+2\log_2\left(\frac{P}{aB}\right)\right]+352a-11a\left(\log_2 B\right) \quad (9)$$

The set-associative PRC costs must also include the cost of the status memory. Assuming a pseudo-LRU replacement policy, the status memory cost is given by Equation 10 as a function of the degree of associativity and the number of sets. Each status bit consumes six transistors for the SRAM cell. Each bit line in the status memory requires 11 transistors for sense amplifiers and write buffers.

$$C_{status} = \frac{P}{aB}6(a\text{-}1)+11(a\text{-}1) \quad (10)$$

The fully-associative design requires the use of content addressable memory (CAM) cells in the instruction tag memory. A CAM cell can be implemented by adding three more transistors to the standard 6-transistor SRAM cell [Ref. 10, p. 589]. Two of these transistors form an XOR gate for comparison and the third is used as a distributed NOR pull-down. All of the bits in the instruction address are used as a tag, resulting in a row size of 30 bits. Equation 11 gives the instruction tag memory cost for a fully-associative PRC.

$$CF_{itag} = \frac{P}{B}(9N)+\frac{P}{B}\left(2\log_2(P/B)+2\right)+11N, \text{ for } N=30$$
$$= \frac{P}{B}\left[272+2\log_2(P/B)\right]+330 \quad (11)$$

The cost of the address predictor, two tag comparators, and two address registers is not included in the cost estimates because they are not expected to contribute significantly to the total cost. The cost of the selection multiplexer used in the set-associative PRC designs is also excluded.

The transistor costs are estimated for nine different PRC sizes and five different associativity choices. The block size is set to 16 bytes in all calculations. Table 4 through Table 8 shows estimated transistor costs for direct-mapped, 2-way set-associative, 4-way set-associative, 8-way set-associative, and fully-associative PRC designs. The total number

of transistors is the sum of the transistor costs associated with the instruction address tag memory (I-Tag), the data address tag memory (D-Tag), the MRMA memory, and the predicted data memory.

PRC Size	I-Tag	D-Tag	MRMA	Data	Total
256	2942	3156	3156	14022	23276
512	5459	6068	6068	26566	44161
1024	10376	11956	11956	51718	86006
2048	19965	23860	23860	102150	169835
4096	38642	47924	47924	203270	337760
8192	74983	96564	96564	406022	674133
16384	145628	194868	194868	812550	1347914
32768	282833	393524	393524	1627654	2697535
65536	549062	794932	794932	3261958	5400884

Table 4. Direct-Mapped PRC Transistor Costs

PRC Size	I-Tag	D-Tag	MRMA	Data	Status	Total
256	3017	3432	3432	15532	59	25472
512	5598	6312	6312	28044	107	46373
1024	10643	12136	12136	53132	203	88250
2048	20488	23912	23912	103436	395	172143
4096	39677	47720	47720	204300	779	340196
8192	77042	95848	95848	406540	1547	676825
16384	149735	193128	193128	812044	3083	1351118
32768	291036	389736	389736	1625100	6155	2701763
65536	565457	787048	787048	3255308	12299	5407160

Table 5. 2-Way Set-Associative PRC Transistor Costs

PRC Size	I-Tag	D-Tag	MRMA	Data	Status	Total
256	3092	4016	4016	18584	105	29813
512	5737	6864	6864	31064	177	50706
1024	10910	12624	12624	56088	321	92567
2048	21011	24272	24272	106264	609	176428
4096	40712	47824	47824	206872	1185	344417
8192	79101	95440	95440	408600	2337	680918
16384	153842	191696	191696	813080	4641	1354955
32768	299239	386256	386256	1624088	9249	2705088
65536	581852	779472	779472	3250200	18465	5409461

Table 6. 4-Way Set-Associative PRC Transistor Costs

PRC Size	I-Tag	D-Tag	MRMA	Data	Status	Total
256	3167	5216	5216	24720	161	38480
512	5876	8032	8032	37168	245	59353
1024	11177	13728	13728	62128	413	101174
2048	21534	25248	25248	112176	749	184955
4096	41747	48544	48544	212528	1421	352784
8192	81160	95648	95648	413744	2765	688965
16384	157949	190880	190880	817200	5453	1362362
32768	307442	383392	383392	1626160	10829	2711215
65536	598247	772512	772512	3248176	21581	5413028

Table 7. 8-Way Set-Associative PRC Transistor Costs

PRC Size	I-Tag	D-Tag	MRMA	Data	Total
256	4810	4660	3156	13990	26616
512	9354	9076	6068	26502	51000
1024	18506	17972	11956	51590	100024
2048	36938	35892	23860	101894	198584
4096	74058	71988	47924	202758	396728
8192	148810	144692	96564	404998	795064
16384	299338	291124	194868	810502	1595832
32768	602442	586036	393524	1623558	3205560
65536	1212746	1179956	794932	3253766	6441400

Table 8. Fully-Associative PRC Transistor Costs

Figure 15 shows the variation in the number of transistors as a function of the PRC size and the associativity. The direct-mapped PRC has the cheapest implementation cost for all PRC sizes. For small PRC sizes, the set-associative designs require more transistors than the fully-associative designs because of the increased row size. Although the number of sets decreases with the degree of associativity for a given PRC size, the transistor cost of sense amplifiers, write buffers, and the word multiplexer increases due to the increased width of each set. However, as the PRC size increases, the transistor cost of the memory cells becomes more dominant in the total cost and the fully-associative designs become more expensive.

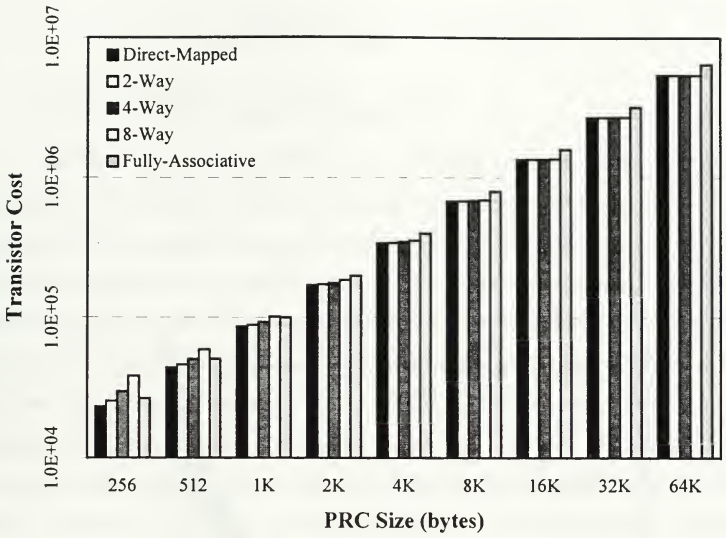

Figure 15. Variation in Transistor Cost

The cost is under 100,000 transistors for 256-byte, 512-byte, and 1-Kbyte PRC sizes. It remains under 1,000,000 transistors for sizes up to 8 Kbytes. Even though the PRC controller and block replacement logic is likely to increase the total cost, a small PRC can easily be integrated on the same chip as the microprocessor using current VLSI fabrication technology.

The difference in the number of transistors between the original and the new PRC costs is plotted in Figure 16. This difference stems from the addition of a new SRAM to hold instruction address tags. The lower part of the plot shows the increase in the number of transistors for direct-mapped PRC designs. The upper part shows the number of transistors required by fully-associative designs in addition to those of direct-mapped designs. The additional cost is less than 150,000 transistors for PRC sizes up to 8 Kbytes. As the PRC size increases, the difference between two PRC designs becomes more significant.

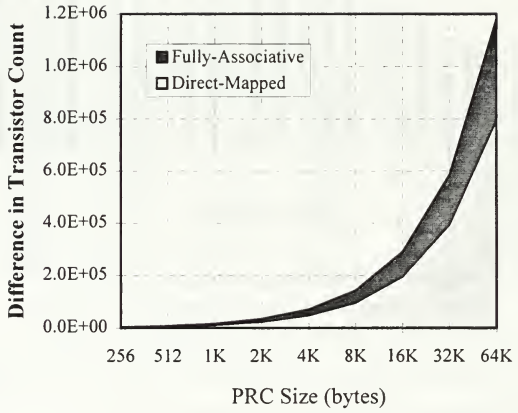

Figure 16. Difference in Transistor Count

IV. ADDRESS TRACE CONVERSION

A. TRACE-DRIVEN CACHE SIMULATIONS

Hardware prototypes, analytical and/or numeric models, address traces, and simulations are the primary tools to evaluate and optimize the cache performance. Numeric models are used in algebraic analysis of cache behavior with moderate accuracy [Ref. 2, p. 14]. Their complexity and applicability range from simple probabilistic models to more sophisticated models based upon measured and derived parameters. Although numeric models give an insight into the characteristics of the cache and the tradeoffs involved in the cache design, they lack practicality because of the assumptions and simplifications made in building the relationships between various parameters. On the other hand, hardware prototypes are the most expensive means of performance evaluation because of the time and resources allocated in developing the prototype hardware. Once the prototype is built, it is either not possible or not very easy to modify the cache organization to test alternative designs [Ref. 11, p. 395]. However, prototype hardware designs can be accurately simulated using analytic models and address traces.

The primary use of simulations in the cache design is to explore the effects of alternative cache characteristics on the system performance before implementing the cache in hardware. Simulations can be classified into two major categories, depending on the source of stimuli used in the simulator: *execution-driven* simulations [Ref. 12, p. 40] and *trace-driven* simulations [Ref. 13, p. 64]. Execution-driven simulations use the capability of the microprocessor to trap to the operating system after the execution of each instruction. A special-purpose trap handler determines what addresses were generated by the instruction and transmits these addresses to the simulator. Trace-driven simulations use external stimuli collected from a microprocessor-based system running a selected workload.

The trace data can be captured by using a monitoring technique implemented in software, hardware, or microcode. Software techniques include inlining (instruction modification), trapping, and emulation. [Ref. 9, p. 443]. Inlining is used to create an

instruction trace by modifying the program at the source, object-code, or executable level. Trapping is very similar to the execution-driven simulations except that the addresses captured are stored in a file, rather than being transmitted to the simulator at run time. Emulation requires running a program to create references according to the behavior of the target architecture. The major disadvantage of these software techniques is the difficulty in capturing the references generated by the operating system. [Ref. 9, p. 443]

Another approach is to perform the tracing below the operating system level by modifying the microcode of the processor. One of the first implementations of this technique is the ATUM (Address Tracing Using Microcode) traces captured from a VAX 8200 processor at Digital Equipment Corporation, Hudson [Ref. 11, p. 396]. All address references generated by the processor are accumulated in a reserved area of the main memory and periodically written out to secondary storage. However, the major drawback of the microcode-based techniques is their confinement to processors with writable or patchable control stores [Ref. 11, p. 396].

Hardware monitoring techniques capture traces directly from the input and output pins of the microprocessor in real time. The specialized hardware that interfaces with the CPU runs several times faster than the host microprocessor in order to sample all signals without any loss of information [Ref. 12, p. 39]. One such technique, BACH (BYU Address Collection Hardware), has been developed at Brigham Young University (BYU) to collect contiguous traces of arbitrary length from three different hardware platforms. BACH can be interfaced with the Intel i80486DX, the Motorola MC68030, and the SPARC microprocessor, running MS-DOS, UNIX SysVR3.2, UNIX SysVR4, Mach2.6, Mach3.0, SUN OS, and HP-UX operating systems. The address traces generated by the BACH system are referred to as BYU traces. [Ref. 9, p. 443]

Miller has used BYU traces from an Intel i80486 platform to simulate the original PRC design [Ref. 8, p. 11]. The new PRC design will also be simulated by using BYU address traces, with the exception that longer traces collected from a SPARC platform will be used. The next section introduces the general aspects of address traces as well as the

BYU trace format. The information needed by the new prediction algorithm and the methods to extract this information from the available address traces are also discussed.

B. ADDRESS TRACES

The accuracy of trace-driven simulations depends on the selection of address traces and whether or not they are representative of the actual workload. Another important factor is the address trace length because simulations running short traces can yield biased results caused by short-term transient behavior of the cache [Ref. 13, p. 65]. However, it is difficult to accumulate long and continuous traces due to the limitations of monitoring hardware. This problem can be overcome either by using a scheme known as trace sampling and stitching [Ref. 11, p. 395] or by temporarily halting the execution of the CPU. The BYU traces are generated with the second technique. The BACH system uses a 6-Mbyte buffer to store the captured data and halts the execution of the CPU while the contents of the buffer are emptied to secondary storage [Ref. 9, p. 445].

On many systems, the operating system dominates the workload and exhibits less locality than user programs. Therefore, exclusion of the operating system from the address traces can cause optimistic performance results [Ref. 11, p. 396]. The input/output activity and the multiprogramming behavior must also be included in the traces. The traces generated by the BACH monitoring system contain all references made by the operating system kernel, multiple user processes, and interrupt routines because they are captured directly from the pins of the microprocessor [Ref. 9, p. 443].

The BYU SPARC traces selected for the PRC simulations are listed in Table 9. Both Kenbus and Sdet are benchmarks from the SPEC SDM (System Development Multitasking) suite [Ref. 14, p. 37]. Each trace name comprises the benchmark name followed by the number of users running the benchmark concurrently. Both benchmarks are characterized by high-frequency UNIX command execution, notably compiler functions, binary executables, random shell scripts, and text processing facilities [Ref. 14, p. 39].

Trace Name	Platform/OS	Number of Files	Compressed Size
Kenbus20	SPARC/ SUN OS	1396	1.26 Gbytes
Kenbus80	SPARC/ SUN OS	1549	2.30 Gbytes
Sdet2	SPARC/ SUN OS	2127	1.96 Gbytes

Table 9. BYU SPARC Traces

Figure 17 shows the binary data structure of the address traces [Ref. 9, p. 451]. Each trace is made of a series of binary files, numbered in the order that they are downloaded from the BACH buffer. The size of a trace file is typically 4.5 Mbytes. Each trace file consists of 12-byte (96-bit) records that are sampled successively by the BACH monitoring hardware. The *Address* field contains the 32-bit virtual address from the CPU address bus, the *Data* field contains the 32-bit data from the CPU data bus, and the *Cycle* field contains the 16-bit cycle count between two consecutive records. The following byte includes six bit fields to store different attributes of a record. The *mio* bit distinguishes between the memory and I/O references. The *wr* bit indicates the type of the reference, being either a read or a write. The *dc* bit is set if the reference is from the data space and cleared if it is from the code space. The 3-bit *size* field holds the size of the reference. Finally, the *sup* bit

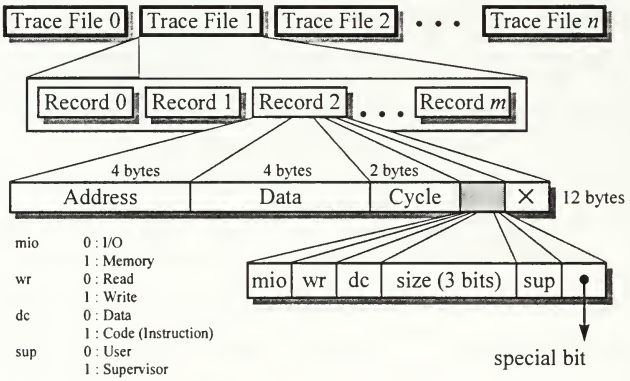

Figure 17. Binary Data Structure of Traces

distinguishes between the supervisor and the user references. The last byte is used as a padding byte and has no significance.

A record can contain three types of references: data, instruction, or special. The data references are ordinary memory transactions for reading from or writing to the memory. The instruction references are instruction fetches for the execution of a program. The special references are used by the designers of the BACH system to insert additional information into the trace file to annotate the occurrence of special events [Ref. 9, p. 454]. When the *special* bit is set, the fields of the record must be interpreted in a different way. The *Address* field contains a number that designates the type of the special reference. The information in the *Data* field depends on the type of the reference. Table 10 lists the special references that are used in the SPARC address traces. These references must be handled very carefully, as explained later in this chapter.

Designator	Description
0	Used as a marker without any special meaning
1	System call
2	Exception
3	Interrupt received
4	Interrupt service routine entered
5	Process ID
6	Access to segment map
7	Access to page map
8	Segment flush
9	Page flush
10	Context flush
11	Integer unit extension
100	Start of a trace segment
101	End of a trace segment
333	Legal reference, whose meaning is unknown
666	Illegal value, or some other error

Table 10. Special References

Although the BYU SPARC traces contain a significant amount of information, they cannot be used directly in the simulations of the new PRC design. These traces do not provide the required relationship between the memory data references and the instructions that create these data references. Therefore, the BYU traces must be converted into a

customized trace format in order to provide the PRC with both data address and instruction address during a memory read cycle. These new traces will be referred to as PRC traces in the rest of this discussion. The next section presents the software tools that are designed to extract the required information from the available BYU traces.

C. SOFTWARE TOOL REQUIREMENTS

The software tools required for the PRC simulations are shown in Figure 18. An address trace editor and a trace conversion program are used together to produce the PRC traces, which are then used to drive a simulator. Since the address trace conversion is an essential part of this research, the conversion tools are introduced prior to the discussion of the design and the operation of the simulator.

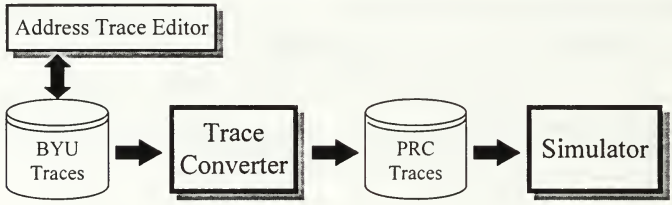

Figure 18. Software Tools

1. Trace Converter (Tracer)

Tracer is a special-purpose program that is designed to convert address traces from the BYU format to the required PRC format. Figure 19 shows the target data structure for the PRC traces, which is slightly different from the BYU trace format. Three of the fields in a PRC record are directly copied from the corresponding BYU record. The 32-bit *Data Address* field is the same as the *Address* field shown in Figure 17, as well as the 16-bit *Cycle* field and the 3-bit *size* field. The *rw* bit, on the other hand, is the complement of the *wr* bit, which indicates whether the reference is a read or write. In addition to the 32-bit data address, a PRC record also contains a 32-bit *Instruction Address* field. In fact, the whole

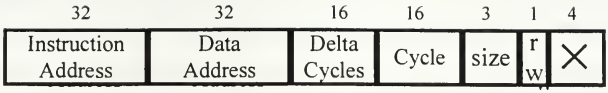

Figure 19. Binary Data Structure for PRC Traces

purpose of the address conversion is to extract the instruction address associated with a data reference from the original traces and to save it together with the data address.

It is important to note that the redundant fields in the BYU format are eliminated by Tracer. A typical cache simulation would only need the memory addresses, not the actual data that is transferred between the CPU and the main memory. Moreover, the simulator does not treat the supervisor and user references differently. Therefore, the *Data* field and the bit fields *mio*, *dc*, *sup*, and *special* are not included in the PRC trace format.

Figure 20 shows the mapping of the references between the BYU and PRC traces. Since the PRC does not interfere with the operation of the instruction cache, the instruction references are not written to the PRC trace file. This does not affect the simulations because Tracer records the number of cycles consumed by the instructions between data references. This information is used by the simulator to determine when the CPU makes a request from the memory subsystem. The total cycle count between consecutive data references is stored into the *Delta Cycles* field of a PRC record. The difference between the *Cycle* field and the *Delta Cycles* field is illustrated in Figure 20. The *Cycle* field contains the cycle count of the

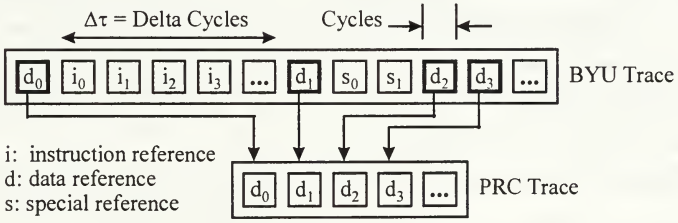

Figure 20. Mapping References from BYU Traces to PRC Traces

data reference itself, while the *Delta Cycles* field contains the total number of cycles since the last data reference. The special references are also left out after they are processed by Tracer.

The conversion algorithm is strongly influenced by the microprocessor architecture that is used to collect the original address traces. The designers of the BACH system has used a SPARCstation 1+ to generate the BYU address traces [Ref. 14, p. 38]. The SPARCstation 1+ complies with the SPARC architecture version 7 specifications [Ref. 15, p. 32]. On the other hand, Tracer employs SPARC architecture version 8, which is upward-compatible from version 7 [Ref. 16, p. xxvi]. The trace conversion program must be modified if address traces are collected using another microprocessor architecture.

Tracer takes advantage of the SPARC memory model, known as *Strong Consistency* or *Strong Ordering*, which requires the loads, stores, and atomic load-stores to be serviced by the memory subsystem in the order that they are issued by the CPU [Ref. 15, p. 84]. Although the SPARC architecture version 8 and 9 specify more advanced memory models to improve the performance, the SPARCstation 1+ uses strong ordering without any instruction prefetches [Ref. 9, p. 452]. The RISC architecture of the SPARC is another advantage because the loads and stores are the only instructions that access memory.

The most prominent obstacle for the conversion process is the distinction between the instruction fetches and the instruction executions. Normally, it would have been very easy to pair each load or store instruction with a corresponding read or write reference according to their sequential order in the trace file. However, the instructions in the BYU traces are not necessarily executed by the CPU, even though they are fetched from the memory. This situation may arise from a number of reasons, such as external interrupts, context switches, branches, subroutine calls, and traps. Therefore, Tracer takes more complicated measures to perform address trace conversions.

Before the operational details of Tracer can be described, a basic understanding is necessary about the control-transfer instructions of the SPARC architecture. The SPARC architecture uses two separate program counters, one for the address of the instruction

currently being executed (PC) and the other for the address of the next instruction to be executed (nPC) [Ref. 16, p. 32]. Table 11 lists five different types of control-transfer instructions which change the value of the next program counter (nPC). These instructions are categorized as *conditional-delayed*, *unconditional-delayed*, and *non-delayed*, with respect to the time at which the control transfer takes place relative to the instruction [Ref. 16, p 50].

The instruction pointed by the nPC during the execution of a delayed control-transfer instruction is referred to as the delay instruction. In general, the delay instruction is the next sequential instruction following the control-transfer instruction, that is, nPC = PC + 4. The use of delay instructions complicates the operation of Tracer, especially when the delay instruction is a load or store. The major problem is introduced by the branch instructions, Bicc, FBfcc, and CBccc, which may or may not execute the delay instruction depending on whether the branch is taken and whether the delay instruction is annulled. The handling of CALL, JMPL, and RETT instructions is relatively straightforward because they always execute the delay instruction unconditionally. Since the trap instructions (Ticc) transfer the control without any delay, they do not affect the operation of Tracer significantly. [Ref. 16, p. 51]

Control-transfer Instruction	Relative Control-transfer Time
Branch (Bicc, FBfcc, CBccc)	conditional-delayed
Call and Link (CALL)	unconditional-delayed
Jump and Link (JMPL)	unconditional-delayed
Return from Trap (RETT)	unconditional-delayed
Trap (Ticc)	non-delayed

Table 11. SPARC Control-Transfer Instructions After [Ref. 16]

The branch instructions are either conditional or unconditional branches. The SPARC architecture specifies a single bit in the opcode of a branch instruction that is used to determine whether the delay instruction is annulled [Ref. 16, p. 119]. Table 12 gives the conditions under which a delay instruction is executed.

As long as the annul bit is cleared, the delay instruction is always executed, regardless of the type of branch. For unconditional branches such as *Branch Always* (BA)

and *Branch Never* (BN), the delay instruction is never executed when the annul bit is set. However, the execution of the delay instruction relies on the result of the conditional branch in case the annul bit is set. If the branch is taken the delay instruction is executed, otherwise it is not executed. [Ref. 16, p. 52]

Annul bit	Branch Type	Branch result	Delay instruction executed ?
a = 0	Conditional	Taken	Yes
		Not Taken	Yes
	Unconditional (BA, BN)	Taken	Yes
		Not Taken	Yes
a = 1	Conditional	Taken	Yes
		Not Taken	No (annulled)
	Unconditional (BA, BN)	Taken	No (annulled)
		Not Taken	No (annulled)

Table 12. Delay Instruction Execution Conditions After [Ref. 16]

The delayed control-transfer of the SPARC architecture is based on the principle that the code can be rearranged by placing useful instructions in the delay slot to maximize the throughput. Thus, the pipeline need not be flushed every time a control transfer occurs [Ref. 15, p. 42]. However, this feature makes it difficult for Tracer to keep track of the instructions that are actually executed by the CPU.

Tracer partitions a trace file into contiguous code segments and operates on each segment individually. A *code segment* is defined as a span of references in which all instructions follow a sequential order. This implies that the difference between the addresses of consecutive instructions must be four bytes because the SPARC architecture uses 4-byte instructions. Although a code segment may also include a number of data and special references, the first reference must always be an instruction reference. If the code segment does not contain any control-transfer instructions, it is called an *inactive code segment* and treated in a different manner by Tracer. However, the code segment cannot have more than one control-transfer instruction. Figure 21 shows the contents of a code segment that is loaded from the trace file in a single conversion cycle.

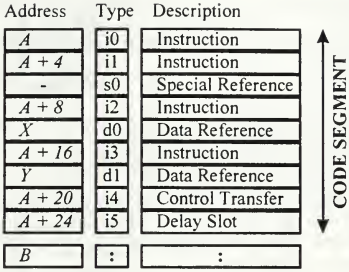

Figure 21. Contents of a Code Segment

A *conversion cycle* consists of two passes over a code segment which eventually creates PRC records from references contained in the segment. Figure 22 shows the functional block diagram of Tracer with three separate buffers to store a code segment.

The *BYU Trace Sorter* provides an interface for reading references sequentially from the BYU traces files. It reads only one code segment in each conversion cycle and sorts references according to their types. Instructions are pushed into the *I-Buffer* and data references are pushed into the *D-Buffer* in the same order as they are read from the trace file. Each time an instruction is read, its address is compared to the address of the previous instruction to make sure that it belongs to the current code segment. The *BYU Trace Sorter* stops loading the *I-Buffer* when it detects the end of a code segment. Then, it pushes the first two instructions from the next code segment into the *B-Buffer*, which is used as a branch buffer to resolve conditional branches. Both the *I-Buffer* and *D-Buffer* may contain a variable number of references depending on the size of the code segment, while *B-Buffer* always contains two entries. The resolution of conditional branches is explained in more detail later in this section.

Tracer makes two passes in a typical conversion cycle to generate PRC traces from the current code segment. The first pass involves only the *I-Buffer*, in which each instruction is decoded by the *SPARC Version 8 Decoder* starting from the first instruction in the buffer. The *Data* field in the BYU trace format contains all 32 bits of the instruction.

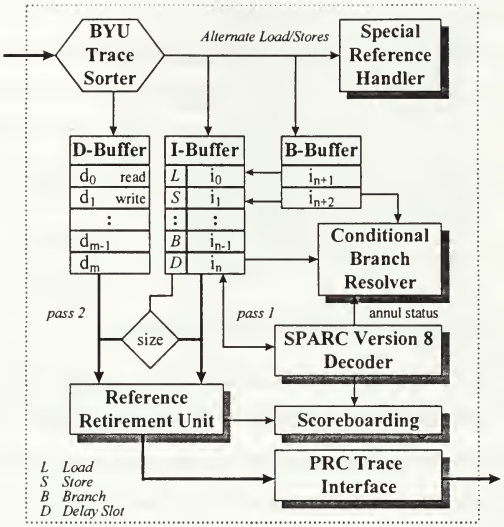

Figure 22. Functional Block Diagram of Tracer

The decoder parses the opcode into the bit-fields specified by the SPARC architecture to find all arguments of an instruction [Ref. 16, p. 44]. There are two types of instructions that are of major interest to Tracer: memory instructions and control-transfer instructions. All other instructions are ignored after they are recorded by the *Scoreboarding* unit.

The first pass is actually a semi-destructive pass, that is, all instructions except loads and stores are removed from the *I-Buffer* as they are decoded. However, whenever a control-transfer instruction is decoded, a number of actions are taken by Tracer before the instruction is removed from the buffer. Tracer first records the type of the control transfer instruction as it is reported by the decoder. If it is a delayed control-transfer instruction, the next entry in the *I-Buffer* is marked as the delay slot. The next action is to determine whether the instruction in the delay slot is executed or not. If the instruction is a CALL, JMPL, or RETT, the delay slot is always executed. If the instruction is a branch, then the

result of the branch must be known as well as the value of the annul bit. This process is extremely important in case the delay slot contains a load or store instruction.

The *Conditional Branch Resolver* determines the result of a conditional branch by using the address of the last instruction in the *I-Buffer*, the address of the last instruction in the *B-Buffer*, and the value of annul bit reported by the decoder. It must be noted that the branch result need not be resolved for unconditional branches BA and BN since it is implicitly contained in the branch instruction itself.

The number of entries needed in the *B-Buffer* to resolve a conditional branch must be chosen according to the branch strategy specified by the architecture and the pipeline structure employed by the implementation. The SPARCstation 1+ uses either the Fujitsu MB86901 or the LSI Logic L64801 microprocessor, which both have a four-stage pipeline [Ref. 15, p. 47]. Tracer takes advantage of the fact that the SPARC architecture assumes taken branches and always fetches the target instruction.

The conditional branch resolution can be explained by using an example. Figure 23 shows a simple assembly language code fragment and its execution by the four-stage pipeline. The pipeline stages are *Fetch*, *Decode*, *Execute*, and *Write* [Ref. 15, p. 60]. The instructions in each stage are shown for clock cycles t_1 through t_5.

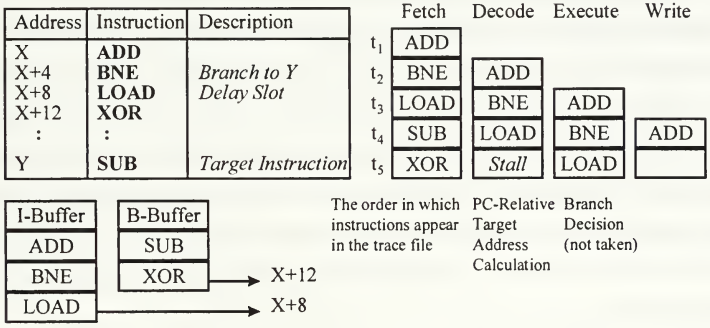

Figure 23. Conditional Branch Resolution

The ADD instruction enters the pipeline in cycle t_1, followed by the BNE (branch not equal) instruction in cycle t_2. The CPU decodes BNE in cycle t_3 as it fetches the LOAD instruction from the delay slot. Since the SPARC branches are PC-relative, the branch target address is also calculated in the decode stage [Ref. 16, p. 120]. The CPU then fetches the target instruction (SUB) instead of the next instruction (XOR) by assuming that the branch will be taken. However, when it is determined that the branch is not taken in the execute stage of BNE, the CPU stalls the pipeline and fetches the XOR instruction in cycle t_5. If the delay slot is annulled, then the LOAD instruction is not executed.

The instructions in the trace file are located in the order that they enter the fetch stage of the pipeline. Tracer pushes the first three instructions, ADD, BNE, and LOAD, into the *I-Buffer*. Since the SUB instruction is not the next sequential instruction following LOAD, Tracer terminates the current code segment and pushes the next two instructions, SUB and XOR, into the *B-Buffer*. It determines the branch result by simply calculating the difference between the addresses of the last entries in each buffer. If the difference is four, then Tracer assumes that the branch is not taken and removes the target instruction (SUB) from the *B-Buffer*. Then, it uses the branch result and the value of the annul bit to determine whether the delay slot (LOAD) is executed according to the conditions given in Table 12. If the delay slot is annulled, then Tracer removes the LOAD instruction from the *I-Buffer*.

When the first pass is completed, the *I-Buffer* contains only load and store instructions that are actually executed by the CPU, while the *D-Buffer* contains all data references loaded from the trace file. The second pass is performed by the *Reference Retirement Unit* which merges the corresponding entries of the *I-Buffer* and *D-Buffer* into the PRC records. The addresses of instructions from the *I-Buffer* are combined with the information contained in the data references starting from the first entry in each buffer. However, both the type and the size of the corresponding entries must be compared to avoid an incorrect conversion.

The SPARC instruction set provides four different sizes for loads and stores: byte (8 bits), halfword (16 bits), word (32 bits), and doubleword (64 bits) [Ref. 15, p. 90]. All

load/store sizes except doubleword result in a single data reference in the trace file because the data bus size is 32 bits. On the other hand, the doubleword loads and stores require two 32-bit data references. Therefore, only a doubleword load/store instruction can retire two adjacent data references from the *D-Buffer*. This implies that the corresponding records in the PRC trace file must contain the same instruction address because their data references are created by the same instruction.

The second pass removes all instruction and data references from the buffers as they are converted and saved into the PRC trace file. However, a data reference associated with a load/store instruction may appear in the next code segment due to the delay slots in the program. Therefore, the *D-Buffer* is never flushed in order to enable the use of remaining data references during the next conversion cycle. The contents of *B-Buffer* is loaded into the *I-Buffer* at the end of a conversion cycle because they belong to the next code segment.

As mentioned earlier, code segments that do not contain a control-transfer instruction are called inactive code segments. These are generally caused by context switches and interrupts. Even though the instructions of an inactive code segment are not used in the conversion, the data references contained in the segment cannot be ignored. If Tracer cannot detect a control-transfer instruction at the end of the first pass, it flushes the *I-Buffer*, retains all data references in the *D-Buffer*, and starts a new conversion cycle.

The last issue in the address trace conversion is the way special references are handled by Tracer. The SPARC architecture uses an 8-bit Address Space Identifier (ASI) appended to the 32-bit memory address to encode the address space being accessed. [Ref. 16, p. 43]. The architecture assigns only four of the 256 identifiers and leaves the remaining assignments to the implementation. The basic address spaces are user instruction, user data, supervisor instruction, and supervisor data [Ref. 16, p. 261]. The alternate spaces can only be accessed by privileged load/store instructions.

Alternate space load/store instructions in BYU traces can easily be distinguished from the ordinary load/store instructions by examining their opcodes. However, the references created by alternate space load/store instructions are not different from the

memory data references. The designers of the BACH system have chosen to encode only four basic address spaces via the bits *sup* and *dc* in order to keep the record size reasonable [Ref. 9, p. 451]. On the other hand, all alternate space data references are annotated with a preceding special reference to provide the necessary ASI value. Therefore, Tracer ignores every reference following a special reference that indicates an access to the alternate space. These special references are listed in Table 10 as integer unit extension, access to segment map, access to page map, segment flush, page flush, and context flush.

2. BACH Address Trace Editor (BATE)

BATE is a command-line application that is written in C++ to interact with the binary trace files created by the BACH monitoring system. It provides a user interface to view and edit the contents of a trace file one reference at a time. It is also used to save trace fragments into an ASCII text file for debugging address conversions performed by Tracer.

The user interface of BATE is shown in Figure 24. A standard three-line header is displayed for all types of references. If the reference is an instruction, then additional information is displayed to show low-level details of the instruction format. BATE uses the same instruction decoder as Tracer and translates the decoder output into the format specified by the SPARC architecture [Ref. 16, p. 44].

```
--------------------------------------------------------------------------------
Reference # : 108         Trace File : input/Sken1.00000    Total  : 375030
Address     : 4161372692  f8098214     INSTRUCTION          Cycles : 1
Data        : 3490111576  d006e058     SUPERVISOR           MEMORY READ [4]
--------------------------------------------------------------------------------

Opcode : DEC = [3490111576] HEX = [d006e058] FORMAT 3
Load Word [LD]

          |op| rd  | op3  | rs1 |i|    simm13    |
Decimal : |03|00008|000000|00027|1|0000000000088|
Binary  : |11|01000|000000|11011|1|0000001011000|

COMMAND> ❑
```

Figure 24. BATE User Interface

The first line of the header contains the attributes of the trace file, such as the current reference number, filename, and total number of references. The next two lines show the fields of the current reference by interpreting the binary data structure given in Figure 17.

The *Address* and *Data* fields are displayed in both hexadecimal and decimal formats. The bit-fields, on the other hand, are converted to meaningful words such as, INSTRUCTION, DATA, SUPERVISOR, USER, MEMORY, I/O, READ, and WRITE. The *size* of the reference is shown in brackets following the other attributes of the reference. BATE also displays an explanation for special references according to the value of *Address* field given in Table 10.

The additional information about instructions includes the opcode, name, format, and the assembly language mnemonic of the instruction. The instruction fields are shown in both binary and decimal formats as they are partitioned by the decoder. These fields provide useful information such as source/destination registers and immediate values. A complete description of instruction fields and formats can be found in Ref. 16.

BATE is designed as an interactive application to run on a request-response basis. It receives a user command from the command-line, performs the required task, and then waits for the next command. The commands recognized by BATE are shown in Table 13.

Command	Shortcut	Arguments	Description
open	o	filename	Opens a trace file with the name filename
next	n	N/A	Displays the next reference
prev	p	N/A	Displays the previous reference
first	f	N/A	Displays the first reference in the file
last	l	N/A	Displays the last reference in the file
goto	g	ref#	Goes to the reference ref#
pos	+	disp	Jumps forward disp references
neg	-	disp	Jumps backward disp references
fnext	>	[d | i | s]	Finds the next reference of specified type
fprev	<	[d | i | s]	Finds the previous reference of specified type
write	w	ref#1 ref#2	Writes from ref#1 to ref#2 into an ASCII file
help	?	N/A	Displays on-line help
Return	N/A	N/A	Repeats the last command
modify	N/A	field [bit#]	Modifies a field or a specified bit of a field
exit (quit)	N/A	N/A	Terminates execution
big	N/A	N/A	Switches to the big-endian mode
little	N/A	N/A	Switches to the little-endian mode

Table 13. BATE Commands

BATE operates on a single trace file at any given time. A trace file can be opened by using the command *open* and providing the filename as an argument. The filename must also include the directory path of the trace file unless the file is located in the current directory.

Most of the BATE commands are used to browse a trace file by providing either an absolute or a relative reference number. The commands *pos* and *neg* use a displacement to jump to a reference relative to the current reference number. The commands *first* and *last* simply set the file pointer to the beginning or end of the file, respectively. The user can also specify an absolute reference number by using the command *goto*. All commands that modify the file pointer assure that the physical file limits are not exceeded. A particular type of reference (*d*: data, *i*: instruction, *s*: special) can be searched by using commands *fnext* and *fprev*.

BATE can write a range of references into an ASCII text file to provide the user with the ability to examine multiple references at the same time. By using the *write* command, the user specifies the first and last reference number of the trace fragment and enters a filename when prompted. Even though the whole trace file can be saved as a text file, it is not recommended because of excessive file sizes. An example output file is shown in Appendix A. Each line in the file represents a single reference. The first column in each line contains the physical reference number. The fields of a reference are written according to the order they are loaded from the binary trace file. All instruction references are tagged with an assembly language mnemonic at the end of the line. Data references are also tagged with the type (read or write) and the size of the transaction. These output files are easier to read than a binary or hexadecimal file for debugging purposes.

The user can also modify the contents of a trace file by using the *modify* command. This command must be used carefully because it changes the value of a reference and updates the binary trace file with the new value. In general, a trace file needs to be modified only if there are some bit errors in the file that might affect an address conversion task substantially. When the user attempts to modify a file, BATE creates a log file with the

name *modify.log* and records each modification into this file with a time stamp. The log file contains both the original and the modified references to enable the recovery from an unintended modification. An example log file is shown in Appendix B.

The commands *big* and *little* are used to inform BATE of the byte ordering convention of the host machine. If the machine supports little-endian ordering, BATE converts address traces from big-endian to little-endian format [Ref. 1, p. E-10].

D. USING CONVERSION TOOLS

The first step in address trace conversions is to run Tracer over a number of BYU trace files. Tracer itself runs quite fast, converting a typical 4.5-Mbyte trace file in less than a minute on a SPARCstation 10. However, messages generated by Tracer during a conversion process must be examined further by the user to ensure that the converted PRC traces are correct. Under normal circumstances, Tracer can successfully convert BYU traces into PRC traces without any problems. Unfortunately, BYU traces can occasionally contain bit errors, attributed to the use of unshielded ribbon cables in the interconnection between various BACH components [Ref. 9, p. 453]. Even though these errors can be tolerated in the simulation of a conventional cache, they may affect the conversion process substantially and distort the outcome of simulations involving the new PRC design. Therefore, the post-processing of errors is an essential part of address trace conversions, taking much more time and effort than the actual conversions performed by Tracer.

All BYU trace files for a given benchmark have the same filename followed by a five-digit file extension that represents the order in which a file is created by the BACH system. Figure 25 shows the basic files involved in the address conversion process. Tracer creates a separate PRC trace file for each BYU trace file used in the conversion. It appends the string "PRC" to the original filename and uses the same file extension as the corresponding BYU trace file.

Tracer is run by using an initialization file (*filename.ini*) containing the input parameters shown in Table 14. The name of the initialization file must be the same as the name of the BYU trace files. The user must provide the common trace name (*filename*) as a

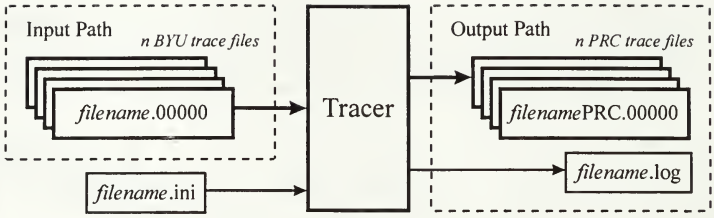

Figure 25. Input and Output Files Used by Tracer

Parameter Name	Description
Input trace path	The directory path for input trace files
Output trace path	The directory path for output trace files and reports
Start file number	The file number to start conversion
Stop file number	The file number to stop conversion
Backup frequency	The frequency with which backups are created
Create log file	Creates a log file for error messages
Create dumps files	Generates ASCII dump files for debugging
Enable e-mail reports	Enables e-mail reports to the user
User e-mail address	User e-mail address to send conversion reports

Table 14. Tracer Input Parameters

command line parameter at the time Tracer is started. Tracer first looks for the file *filename.ini* in the current working directory and reads the required input parameters from this file.

The first two parameters specify the input and output directory paths for BYU and PRC traces. The input path must point to the directory in which BYU trace files are located. The output path is used by Tracer to save the converted PRC trace files and a log file with the name *filename.log*. The conversion errors and warnings are recorded into the log file rather than being displayed at the standard output device in order to enable background jobs.

Tracer uses the start and stop file numbers to determine the set of trace files to be converted. A single file can be converted by setting these numbers to the same value. Tracer is designed to convert a series of trace files in an arbitrary number of runs. For example, the

files from *filename.00000* to *filename.00099* can progressively be converted in groups of 10 files by specifying the first and the last file extension in each run. However, the logical order of files cannot be changed because Tracer continues the conversion by using the previous conversion results.

At the end of a conversion task, Tracer creates a number of binary files to save the contents of instruction and data buffers as well as the values of internal variables. These files are given the same extension as the last BYU trace file converted by Tracer. If the start file number is not equal to zero, i.e., there must have been a previous conversion, then Tracer looks for the saved binary files with an extension one less than the current start file number. The names of these binary files are "*INSTsave*", "*DATAsave*", and "*save*". The conversion is aborted if these files are missing from the output directory. This progressive conversion approach provides the flexibility to divide a large set of trace files into easily manageable groups.

Tracer can also create ASCII dump files for each binary file it saves. These files are very useful in debugging Tracer and obtaining statistical information about the traces. The backup frequency specifies how often Tracer should create binary and ASCII backup files during a conversion. If the backup frequency is set to *f*, then Tracer creates backups every *f* trace files by using the file extension of the current file. If the backup frequency is set to zero, no backups are created until the end of the conversion.

The post-processing phase of the conversion involves examining the Tracer log file and making necessary corrections to BYU traces by using BATE. The error messages that can be reported by Tracer are shown in Table 15. Most of the errors are related to the discrepancies between the instruction and data references during the retirement pass of a conversion cycle. It must be noted that only some of these errors affect the conversion process significantly. These errors generally cause a chain reaction in which all of the following retirements shift one or more references.

Error Message	Description
Size Mismatch	Data reference size is different than the instruction size
Read Mismatch	Data reference type is different than the instruction type
Orphan Read/Write	There is no parent load/store for a read/write reference
Cycle Conflict	There is only one data reference for a doubleword load/store
Memory-indirect load	There is only one read reference for two load instructions

Table 15. Tracer Error Messages

Size and read mismatches are generally caused by single-bit errors in the opcode of load/store instructions. A single-bit error can turn a load-byte instruction into a load-doubleword instruction, or vice versa [Ref. 16, p. 90]. However, sometimes the error stems from the data reference itself rather than the instruction opcode. Single- or double-bit errors in the *size* field of a data reference can cause a size mismatch between the data reference and the corresponding load/store instruction. Therefore, it is very difficult to detect the real cause of these errors at runtime.

The orphan read/write error occurs when a data reference does not have a parent load/store instruction in the trace file. These errors are usually detected in the beginning of a trace file and are not very common.

The cycle conflicts arise from a situation in which a double-cycle load/store has a single-cycle data reference to match with. Some cycle conflicts can be the side effect of a size mismatch as explained above. However, most of these errors are caused by external interrupts or context switches. For example, if an interrupt or context switch occurs right after the first cycle of a doubleword load, the second cycle is never completed. These errors can be resolved by Tracer at runtime.

The memory-indirect load is not an error but a warning issued by Tracer. It indicates that two atomic load instructions are interpreted as a single memory-indirect load instruction.

Tracer also reveals useful statistics about the distribution of references in address traces and the instruction mix of the benchmarks. The first 100 files from each benchmark are converted for PRC simulations and the results are given in Table 16.

Benchmark: (100 files each)	Kenbus20	Kenbus80	Sdet2
Total number of BYU references	37,217,400	37,414,516	44,034,770
Total number of PRC references	6,988,242	7,121,893	8,181,576
Conversion ratio	18.777%	19.035%	18.580%
Number of instruction fetches	29,523,576	29,126,515	34,577,140
Number of memory data reads	4,780,418	4,901,106	5,628,188
Number of memory data writes	2,208,496	2,221,822	2,554,753
Number of alternate space reads	34,334	41,678	42,196
Number of alternate space writes	316,976	539,349	558,405
Number of special references	353,587	583,976	640,313
Number of supervisor references	22,291,090	34,470,484	35,212,760
Number of user references	14,221,400	1,778,959	7,547,321
Instruction count	22,744,796	22,288,177	26,330,114
CALL, JPML, RETT	1,050,455	1,138,318	1,375,076
Conditional branches	3,929,653	3,708,064	4,407,202
Unconditional branches	412,377	415,902	549,752
Trap instructions	30,396	24,134	23,205
Integer instructions	10,844,089	10,451,355	12,371,116
Alternate space load/stores	301,376	515,795	562,215
Load byte (LDSB, LDUB)	981,264	718,845	933,893
Load halfword (LDSH, LDUH)	192,282	217,533	210,838
Load word (LD)	2,925,208	2,955,090	3,482,646
Load doubleword (LDD)	340,626	504,525	500,123
Total number of loads	4,439,380	4,395,993	5,127,500
Store byte (STB)	337,461	204,721	203,731
Store halfword (STH)	74,946	105,033	112,008
Store word (ST)	852,602	745,282	957,695
Store doubleword (STD)	471,662	583,281	640,364
Total number of stores	1,736,671	1,638,317	1,913,798
Total number of load/stores	6,176,054	6,034,311	7,041,299

Table 16. Address Trace Statistics Generated by Tracer

The resulting PRC references are about 19% of the original BYU references due to the elimination of instruction and special references. Figure 26, Figure 27, and Figure 28 show the frequency of references for the Kenbus20, Kenbus80, and Sdet2 benchmarks, respectively. The instruction references dominate all three address traces with a frequency of almost 80%. The combined frequency of special and alternate space data references is about 2% to 3% depending on the supervisor activity in each trace category.

Figure 29 shows the distribution of supervisor and user references for all trace categories. Supervisor instruction fetches constitute 48%, 76%, and 66% of the total memory references for Kenbus20, Kenbus80, and Sdet2, respectively. The impact of the

multitasking workload on the supervisor activity can be seen by comparing Kenbus20 and Kenbus80 benchmarks. Since Sdet2 benchmark employs a more strenuous test of operating system commands, its supervisor activity is close to that of Kenbus80, even with two users. The user data references make up 6.4%, 0.7%, and 2.9% of the total references for Kenbus20, Kenbus80, and Sdet2, respectively.

The instruction mix of the benchmarks are given in Figure 30. Tracer calculates the instruction mix by using the instructions actually executed instead of the instructions fetched. A comparison between the number of instruction fetches and the instruction count in Table 16 suggests that 76% of the fetched instructions are executed by the CPU. The instruction mixes of the benchmarks are quite close to each other so that about 50% of the instructions are integer operations. The other 50% are equally distributed between the control transfer and load/store instructions. However, the number of loads is almost three times the number of stores. The number of alternate space instructions is not significant. Since these benchmarks do not contain any floating-point or coprocessor instructions, they are not included in Figure 30.

Figure 31 shows the distribution of CALL, JMPL, RETT, unconditional branch, and conditional branch instructions for each benchmark. More than 70% of the control-transfer instructions are conditional branches. The distribution is consistent among all benchmarks. The number of trap instructions is so small that that it cannot be seen in the graph.

Finally, Figure 32 and Figure 33 show the distribution of loads and stores, respectively, in terms of the instruction sizes. The word-size instructions make up more than 50% of both loads and stores. The doubleword stores are more frequent than doubleword loads.

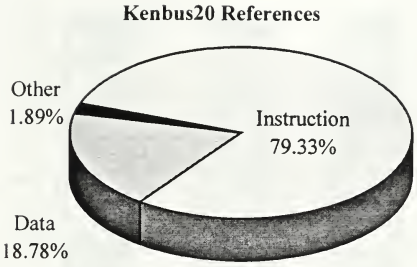

Figure 26. Frequency of References in Kenbus20 Traces

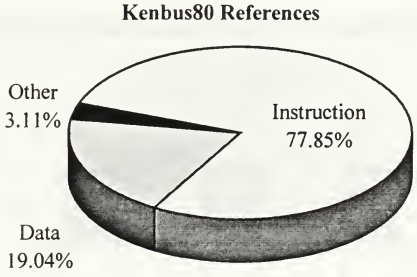

Figure 27. Frequency of References in Kenbus80 Traces

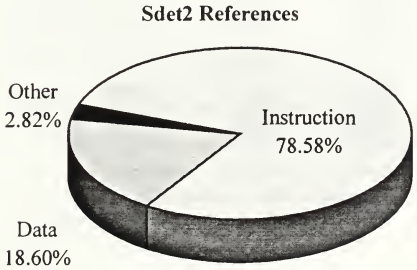

Figure 28. Frequency of References in Sdet2 Traces

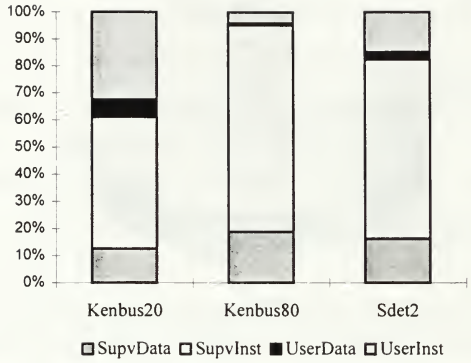

Figure 29. Distribution of Supervisor vs. User References

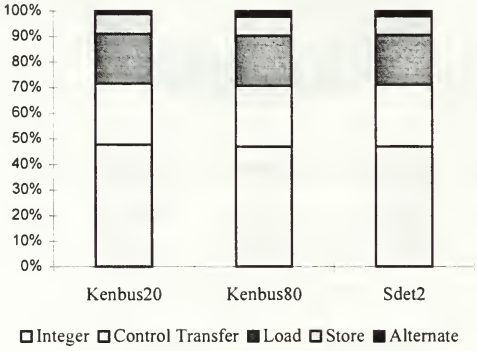

Figure 30. Instruction Mix

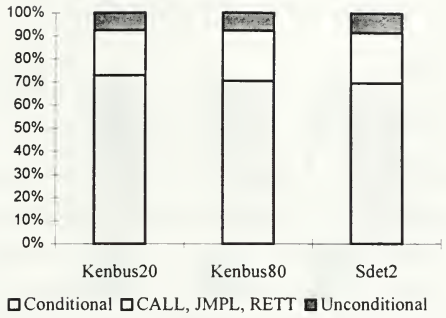

Figure 31. Distribution of Control-Transfer Instructions

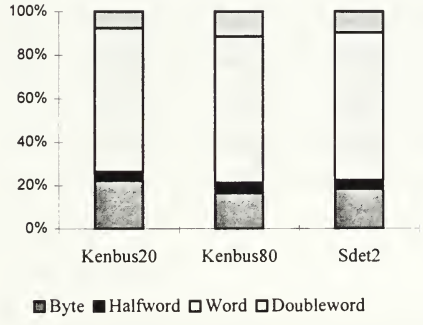

Figure 32. Distribution of Load Instructions

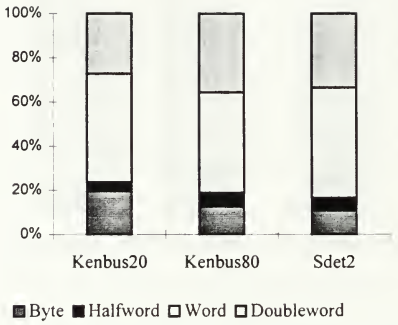

Figure 33. Distribution of Store Instructions

V. CACHE AND PRC SIMULATOR

A. INTRODUCTION

The Cache and PRC Simulator (CaPSim) is a multi-level memory hierarchy simulator that incorporates both statistical and temporal analyses for a user-defined memory subsystem. It allows the designer to evaluate the overall system performance with respect to the organizational parameters and policies selected at each memory level.

A top-down approach has been taken in the design of CaPSim with three primary objectives: portability, flexibility, and efficiency. The CaPSim source code has been written in the object-oriented C++ programming language by using standard libraries and avoiding platform-dependent code. The portability of the source code across different platforms is especially important in simulations using platform-dependent address traces or in execution-driven simulations using run-time stimuli.

Another aspect of CapSim is its flexibility to simulate an unlimited number of combinations in the memory hierarchy without the need for recompilation. CapSim can be configured at run-time to simulate a wide range of cache and PRC designs by using a simple simulation language.

In general, the design of a simulator requires a trade-off between flexibility and efficiency. Efficiency should not be ignored because the trace-driven simulations make intensive use of the CPU and the I/O subsystem. Before starting a simulation, CaPSim uses host computer resources abundantly in order to provide flexible configuration options to the user. However, once the actual simulation starts, CaPSim allocates all resources effectively to improve the efficiency of the simulation.

B. SOFTWARE ARCHITECTURE

The software architecture of CaPSim is based on five fundamental classes, called simulation modules, which represent the behavioral abstractions of different hardware entities used in the memory hierarchy. Figure 34 shows the major building blocks of the

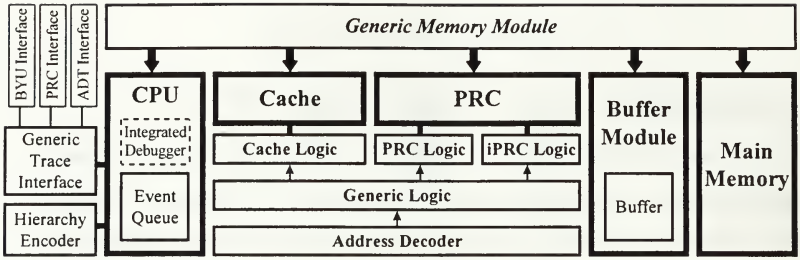

Figure 34. Architecture of CaPSim

CaPSim architecture, including the simulation modules *CPU*, *Cache*, *PRC*, *Buffer Module*, and *Main Memory*. All of these blocks are implemented with a separate C++ class that encapsulates the data variables and methods (functions) associated with the functionality of the block. The simulation modules are derived from an abstract base class called the *Generic Memory Module* to provide a standard interface for inter-module communications and to minimize the interdependency among simulation modules.

The *CPU* class primarily emulates the way the CPU generates requests to the memory subsystem. It uses the *Generic Trace Interface* class to obtain the information required to make a request. The low-level details of reading a trace record from a trace file are isolated from the *CPU* class by using three separate trace interfaces, namely the *BYU Interface*, *PRC Interface*, and *ADT (ASCII Debug Trace) Interface*. All three interfaces are derived from the *Generic Trace Interface* class that standardizes the information passed to the CPU.

The *CPU* class also contains the *Event Queue* class that sorts the events reported by the simulation modules (including the CPU itself). Each event is sorted according to the simulation time at which it is reported to occur. CaPSim increments the system clock by using the reported time of the closest event instead of incrementing it by one cycle in each iteration of the simulation. This improves the simulation efficiency because the CPU spends most of its time executing instructions rather than making memory requests.

Although CaPSim can simulate any memory hierarchy, the programmer is given the flexibility to limit the number of hierarchy options that can be defined by the user. The *Hierarchy Encoder* class is used by the *CPU* class to determine whether the memory hierarchy defined by the user is valid or not. Thus, the user cannot specify irrational combinations of simulation modules in the memory hierarchy, such as two main memories or a cache memory that is located upstream of the main memory.

The *Cache* class emulates the behavior of a cache memory depending on the organizational parameters and policies defined by the user. It uses a state machine to simulate various phases of a cache transaction while leaving the low-level details to the *Cache Logic* class. The *Cache Logic* class is derived from the *Generic Logic* class which contains the basic memory arrays of a cache, such as the tag memory, valid bits, dirty bits, and replacement status bits. It also inherits the *Address Decoder* class that maps a memory address into a cache block.

The *PRC* class is quite similar to the *Cache* class except that it emulates a predictive read cache instead of a conventional cache. It uses two separate logic interfaces, one for the original PRC with data address tags and the other for the new PRC with instruction address tags. However, as far as the other simulation modules are concerned, the behavior of the PRC is the same for both designs. The *PRC Logic* class contains two more memory arrays in addition to those inherited from the *Generic Logic* class. These memory arrays are used to implement the displacement-based prediction algorithm of the original PRC. On the other hand, the *iPRC Logic* class uses an additional memory array to store the instruction address tags. Although these two classes implement different prediction algorithms, they provide the same interface to the *PRC* class.

The *Buffer Module* class is used to emulate the read and write buffers between consecutive levels of the memory hierarchy. Normally, these buffers could be placed inside the *Cache* and *PRC* classes because they are the only simulation modules that may use some kind of buffering. However, it would cause a dependency between these two simulation modules because the first-level data cache and the PRC use the same read and

write buffers [Ref. 8, p. 9]. Therefore, these buffers are implemented in the *Buffer Module* class that can logically be inserted between a cache (or a PRC) and the next level in the hierarchy. The *Buffer Module* class uses the *Buffer* class to dynamically create the read and write buffers of specified sizes. This also gives the user the flexibility to define only a read or write buffer as well as a combination of the two.

The *Main Memory* class is the simplest simulation module that emulates the access and transfer phases of a main memory transaction. It does not require any array allocation in the memory and sinks all requests from the downstream memory levels in the order they are issued. All main memory transfers are implemented as burst mode transfers [Ref. 5, p. 69].

CaPSim is designed with as much potential as possible to allow the integration of additional simulation modules into the source code in future versions. The source code of CaPSim does not contain any global variables in order to minimize the dependencies between individual classes. If the programmer wants to extend the memory hierarchy beyond the main memory level, he or she can incorporate new classes into CaPSim to emulate the components of a virtual memory system, such as a translation lookaside buffer (TLB) or disk subsystem. This requires only minor modifications in the *CPU* class and a compliance with the interface provided by the *Generic Memory Module*. The programmer can also incorporate a new trace interface to drive CaPSim with a different type of address trace without any modification in the simulation modules. CaPSim can even be turned into an execution-driven simulator by implementing a run-time interface instead of a trace file interface. The *CPU* class uses standard function calls to obtain the information associated with a request without any knowledge about the source of the information.

CaPSim defines four primitive data types in addition to those provided by C++ libraries: *String*, *Boolean*, *Clock*, and *MemoryTransaction*. The *String* class replaces the standard C++ string type to give CaPSim more flexibility and power in string manipulations. The *Boolean* class is used in handling boolean values which are normally implemented as integers in C++. The *Clock* class implements a linear timer that is used by

simulation modules to keep track of simulation time. It checks the value of the timer against overflows and underflows each time it is incremented or decremented.

The *MemoryTransaction* class contains the information that is passed between simulation modules in a memory transaction. It also implements a number of overload operators to manipulate the transaction contents.

The source code of CaPSim includes a total of 59 files (26 header files and 23 source files).

C. OPERATIONAL DETAILS

The operation of CaPSim proceeds in three distinct phases: configuration, simulation, and evaluation. Figure 35 illustrates the execution order of the CaPSim phases in which the output of one phase is used as an input for the following phase. The configuration and evaluation phases generally take a few seconds to execute. Most of the execution time is spent in the simulation phase. This section describes the operational details and particular classes related to each phase.

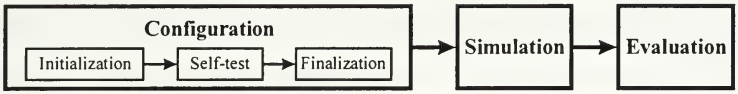

Figure 35. Operational Phases of CaPSim

1. Configuration

CaPSim is invoked by providing an arbitrary configuration filename at the command-line. It searches for the configuration file in the current working directory by appending the file extension "*.cfg*" to the filename specified by the user. The execution is started from the *main()* function which statically creates an instance of the *CPU* class and initiates the configuration phase.

The configuration phase comprises three sub-phases: initialization, self-test, and finalization. The *Generic Memory Module* declares three virtual functions associated with

each sub-phase, as shown in Table 17. All simulation modules override these functions to execute their own customized tasks related to the configuration phase.

Function signature	Description
Boolean initialize (*String&*)	Used for parameter initialization in all modules except the CPU
Boolean selfTest ()	Used for range-checking on the input parameters
Boolean finalize ()	Used for consistency-checking and dynamic allocation of the data

Table 17. Configuration Functions of Simulation Modules

Normally, the initialization function is used for passing input parameters from the configuration file to the corresponding simulation modules by making a separate call for each parameter. However, the initialization function of the *CPU* class is implemented in a different way than those of the other simulation modules. It is called only once by the *main()* function, with a string parameter containing the name of the configuration file. The rest of the simulation modules are initialized by the CPU as they are dynamically allocated.

The actions taken by the *CPU* class during the configuration phase are demonstrated with a flowchart in Figure 36. The shaded blocks indicate that the process is performed repeatedly for each simulation module specified by the user. The initialization phase is followed by the self-test and finalization phases after the simulation modules are allocated and initialized.

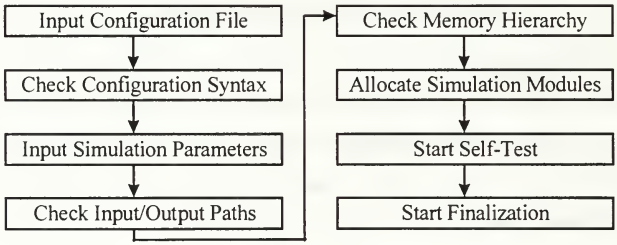

Figure 36. Actions Taken by the CPU During Configuration

CaPSim uses a simple simulation language to interpret the user-defined input parameters and configure the simulation modules. Table 18 lists the reserved keywords of the CaPSim simulation language and Figure 37 shows the syntax of a sample configuration file. The keywords are all case-sensitive.

Keyword	Description	Scope
simulation	Precedes the block in which simulation parameters are defined	file
hierarchy	Precedes the block in which the simulation modules are declared	file
module	Precedes a block that contains parameter definitions for a declared module	file
cache	Used to declare a cache module within the hierarchy block	hierarchy
prc	Used to declare a PRC module within the hierarchy block	hierarchy
buffer	Used to declare a buffer module within the hierarchy block	hierarchy
memory	Used to declare a main memory within the hierarchy block	hierarchy

Table 18. Keywords Used by the CaPSim Simulation Language

All user comments start with the pound sign '#' and continue until the end of the line. CaPSim reads the contents of the configuration file line by line into a string buffer by marking all comments and null lines. It then performs a syntax check by using the information in the string buffer.

The keywords *simulation*, *hierarchy*, and *module* must be followed by a block enclosed with braces, '{' and '}.' Each brace occupies a single line without any other characters to simplify the parsing process. However, the position of the brace in the line is not important because the white-space characters (spaces and tabs) are trimmed by CaPSim. The blocks can be defined in any order within the configuration file. During the syntax check, CaPSim ensures that the file contains only one *simulation* block and one *hierarchy* block. All blocks are enclosed within braces. However, block contents are not examined at this step.

Once the syntax of the file is confirmed, the simulation parameters are initialized by using the information contained in the *simulation* block. The parameter names are not case-sensitive and the number of white-space characters within a parameter name is immaterial. CaPSim compresses the parameter names into single words and converts each character to lower-case. Therefore, the strings "Start File Number," "start FILEnumber," and

```
# ------------------------------------------------------------
# Sample CaPSim configuration file
# ------------------------------------------------------------

simulation
{
   Input  Path          = input/        # Path for trace files
   Output Path          = output/       # Path for output files
   Trace Type           = BYU
   Trace Filename       = Ssdet.*****  # Five-digit file extension
   Start File Number    = 0             # First file extension
   Stop  File Number    = 99            # Last  file extension
   Trace Buffer Size    = 2000
   Word Size            = 4             # Number of bytes per word
   User E-mail Address = fnaltmis@nps.navy.mil
}

hierarchy
{
   cache  CacheL1
   prc    PRC
   buffer Buffers
   memory MainMemory
}

module CacheL1
{
   Parameter name = Value
   ...
}

module Buffers
{
   Parameter name = Value
   ...
}

module MainMemory
{
   Parameter name = Value
   ...
}

module PRC
{
   Parameter name = Value
   ...
}
```

Figure 37. Configuration File Syntax

"startfilenumber" represent the same parameter. Each parameter definition must be located on a single line with an equal sign between the parameter name and the corresponding parameter value.

Simulation parameters define a number of trace file attributes, such as the trace type, the filename, and the extensions of the first and last trace files. The number of digits in the file extension is determined by the number of asterisk characters in the filename. For example, in Figure 37, the filename "Ssdet.*****" and the file extension 99 are translated into "Ssdet.00099." However, the user may also specify a single trace file by eliminating the asterisk characters from the filename. In this case, the start and stop file numbers are ignored by CaPSim.

The input path specifies the directory path in which the trace files are located. The output path is used by CaPSim to create output files at the end of a simulation. Both path names are optional and the default path is the current working directory from which CaPSim is invoked. CaPSim checks whether the input and output paths exist before proceeding with the initialization.

The next step in the initialization phase is the validation of the memory hierarchy specified by the user. CaPSim provides four keywords for the declaration of the simulation modules: *cache*, *prc*, *buffer*, and *memory*. These keywords are recognized only in the scope of the *hierarchy* block. Each module must be declared at a separate line by specifying a unique module name following the keyword. The configuration file must contain exactly one *module* definition for each module declared in the *hierarchy* block. A *module* block must be defined by using the same name as used in the declaration. CaPSim will ignore any extra *module* blocks that do not have a corresponding declaration.

The order of the simulation modules is tested by using the *Hierarchy Encoder* class. Each line in the *hierarchy* block represents a distinct memory level and the first line contains the closest simulation module to the CPU. The *Hierarchy Encoder* class encodes the memory hierarchy into an unsigned integer number by using a different coefficient for each module type. It also contains an array of valid hierarchy codes predefined by the

programmer. If the encoded hierarchy code has a match in this array, then the memory hierarchy is considered as valid. Each row in Table 19 shows a valid hierarchy that can be declared by the user in the current version of CaPSim.

Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
memory					
cache	memory				
buffer	memory				
cache	buffer	memory			
prc	buffer	memory			
cache	cache	buffer	memory		
cache	prc	buffer	memory		
cache	prc	buffer	cache	memory	
cache	buffer	cache	buffer	memory	
cache	buffer	prc	buffer	memory	
cache	prc	buffer	cache	buffer	memory
cache	buffer	cache	prc	buffer	memory

Table 19. Valid Hierarchy Declarations

The simulation set of CaPSim is implemented as an array of pointers that stores the base addresses of all simulation modules. The size of the simulation set is determined by the number of declarations in the *hierarchy* block. However, the *CPU* is always the first module in the simulation set, even though it is not declared explicitly.

The *SimulationSet* array is declared in the *Generic Memory Module* class and shared among all simulation modules. The array itself is created dynamically, depending on the size of the simulation set. The *CPU* first inserts its own base address into the array and then dynamically allocates other modules in the order of declaration. After a module instance is created, the *CPU* calls its initialization function several times until all input parameters in the corresponding *module* definition are passed to the module instance. However, it must be noted that the *CPU* has no knowledge about the significance of these input parameters. It simply passes each parameter to the related simulation module by using a string argument. Each string is then parsed and interpreted by individual simulation modules and the result is reported to the *CPU* through a boolean return type. This approach avoids a dependency between the *CPU* and the other simulation modules. The programmer can add a new input parameter to a particular simulation module without any modification in the *CPU* class.

The *CPU* passes two parameters to simulation modules during dynamic allocation. The first parameter is a string that contains the user-defined module name. The second is a pointer to the system clock (*const Clock&*). However, the system clock is passed as a constant reference in order to prevent simulation modules from modifying the system clock value. The system clock in CaPSim can be read by all simulation modules but modified only by the *CPU* class. The *CPU* makes two more function calls to set the *module ID* and *slave ID* of a simulation module immediately after its allocation. These identification numbers are used as indices into the *SimulationSet* array.

The input parameters can be defined in any order within a *module* block. The parameter names recognized by the *Main Memory*, *Buffer Module*, *Cache*, and *PRC* classes are given in Table 20, 21, 22, and 23, respectively. The optional parameters are given default values by CaPSim if they are not defined by the user. The parameter type indicates the set of values that can be assigned to a parameter. The usage of the input parameters will be explained in the simulation phase discussed later in this chapter.

Parameter Name	Parameter Type	Default Value
Access Time	unsigned integer	*required*
Transfer Time	unsigned integer	*required*

Table 20. Main Memory Input Parameters

Parameter Name	Parameter Type	Default Value
Read Buffer Size	unsigned integer	*required*
Write Buffer Size	unsigned integer	*required*
Write Buffer Block Size	unsigned integer	*required*
Enforce Priorities	[Yes, No] or [True, False]	Yes
Remove Read Duplicates	[Yes, No] or [True, False]	Yes
Remove Write Duplicates	[Yes, No] or [True, False]	Yes
Search Read Buffer	[Yes, No] or [True, False]	Yes
Search Write Buffer	[Yes, No] or [True, False]	Yes

Table 21. Buffer Module Input Parameters

Parameter Name	Parameter Type	Default Value
Cache Size	unsigned integer	*required*
Block Size	unsigned integer	*required*
Sub-block Size	unsigned integer	Block Size
Fetch Size	unsigned integer	Block Size
Transfer Size	unsigned integer	Sub-block Size
Associativity	unsigned integer, * (fully-associative)	*required*
Replacement Policy	FIFO, LRU, Random, Pseudo-LRU	*required*
Write Policy	Write Through, Write Invalidate, Write Update, Write Back	*required*
Write Miss Policy	Write Allocate, Write Around	*required*
Wrapping Fetch Policy	Wrap Up, Wrap Down	Wrap Up
Read Access Time	unsigned integer	*required*
Write Access Time	unsigned integer	*required*
Read Hit Time	unsigned integer	0
Read Miss Time	unsigned integer	0
Write Hit Time	unsigned integer	0
Write Miss Time	unsigned integer	0
Block Buffer Transfer Time	unsigned integer	*required*
Enable Block Buffer	[Yes, No] or [True, False]	No
Search Block Buffer	[Yes, No] or [True, False]	No
Read Forward	[Yes, No] or [True, False]	No
Read Priority	unsigned integer	*implicitly set*
Write Priority	unsigned integer	*implicitly set*
Write Allocate Priority	unsigned integer	*implicitly set*
Write Back Priority	unsigned integer	*implicitly set*

Table 22. Cache Input Parameters

After the simulation modules are allocated and the input parameters are successfully initialized, CaPSim performs a self-test and finalization for each module. The self-test phase involves a range-check on the initialized input parameters. The required parameters are also checked to ensure they are defined by the user.

The optional parameters are assigned default values in the finalization phase, if they are not defined in the configuration file. The consistency among input parameters is also tested in this phase. For example, if the block size of a cache memory is greater than the cache size, then a consistency error will occur during the finalization. The last part of the finalization phase involves the derivation of internal variables from the input parameters and the allocation of sub-classes in the memory. CaPSim allocates all classes and arrays conditionally in order to utilize the available memory in the best possible way.

Parameter Name	Parameter Type	Default Value
Prediction Algorithm	Instruction Address Displacement, Data Address Displacement	Data Address Displacement
PRC Size	unsigned integer	*required*
Block Size	unsigned integer	*required*
Sub-block Size	unsigned integer	Block Size
Fetch Size	unsigned integer	Block Size
Transfer Size	unsigned integer	Sub-block Size
Associativity	unsigned integer, * (fully-associative)	*required*
Replacement Policy	FIFO, LRU, Random, Pseudo-LRU	*required*
Write Policy	Write Through, Write Invalidate, Write Update	*required*
Read Access Time	unsigned integer	*required*
Write Access Time	unsigned integer	*required*
Read Hit Time	unsigned integer	0
Read Miss Time	unsigned integer	0
Write Hit Time	unsigned integer	0
Write Miss Time	unsigned integer	0
Block Buffer Transfer Time	unsigned integer	*required*
Bypass Write Allocates	[Yes, No] or [True, False]	Yes
Minimum read size in buffer	unsigned integer	*disabled*
Maximum read slips in buffer	unsigned integer	*disabled*
Read Priority	unsigned integer	*implicitly set*

Table 23. PRC Input Parameters

All messages generated during the configuration phase are saved in a log file with the same name as the configuration file. CaPSim creates the log file in the current working directory with a file extension of "*.log.*" An example log file is shown in Appendix C.

2. Simulation

The simulation phase starts with a call to the *startSimulation()* function of the *CPU*, which contains the main-event loop of the simulation. It continues until all user-specified trace files are processed by the trace interface. There are two types of events in a CaPSim simulation, internal and external. External events are caused by either requests from downstream modules or responses from upstream modules. Internal events, on the other hand, are triggered by the state machine of a simulation module fulfilling the timing requirements of a transaction. For example, when the CPU makes a read request from the first-level data cache, a read access starts due to an external event. Then, the cache generates a number of time-dependent internal events until the CPU request is serviced.

Figure 38 shows the flowchart of the main-event loop and its interaction with the *Event Queue*. The main-event loop is entered after the *CPU* makes the first request in the simulation. The loop can only be terminated by the *CPU* when the trace interface reports the end of the last trace file in the current simulation. The entries in the *Event Queue* consist of a *module ID* and a *target time* at which the event is reported to occur. The inner loop (shadowed block in Figure 38) retires all events from the *Event Queue* with target times that are equal to the current value of the system clock. After all pending events are evaluated, the system clock is adjusted to the target time of the next entry in the *Event Queue* and a new iteration is executed.

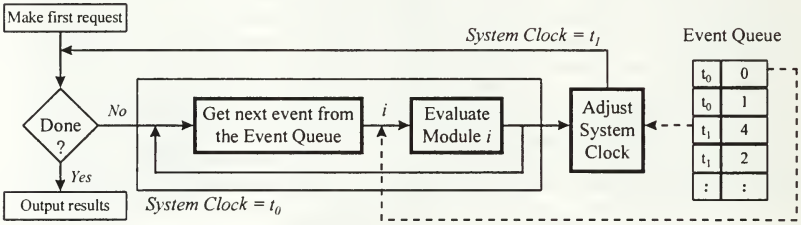

Figure 38. Main Event Loop

The *Generic Memory Module* provides three virtual functions for inter-module communications. The signatures of these functions are shown in Table 24. The *request* function is used to make a memory request from the slave module and the *respond* function is used to make a response to the master module. Both functions take a constant *MemoryTransaction* argument that contains the attributes of a memory transaction. In addition, the *respond* function also takes a constant *Clock* argument that indicates the finish time of a transaction. All arguments are implemented as constant references in order to avoid the overhead of passing parameters by their values. On the other hand, the *cancel* function is used to cancel the last request made to the slave module.

HandShakeType **request** (const *MemoryTransaction&*)
HandShakeType **respond** (const *MemoryTransaction&* , const *Clock&*)
HandShakeType **cancel** ( )

Table 24. Inter-Module Communication Functions

The handshake between two simulation modules is implemented with an enumeration type called the *HandShakeType*. Each inter-module communication function returns a handshake message to the calling module in order to report the result of the transaction. Possible handshake messages are *acknowledge*, *busy*, and *error*. If a simulation module can service a request immediately, it returns *acknowledge*. Otherwise, it returns *busy* to indicate that the request cannot be serviced at the current simulation time. The *error* message is returned only if the function call is illegal. This situation may result from a call to the *request* function of the *CPU* class or to the *respond* function of the *MainMemory* class.

CaPSim takes advantage of two basic object-oriented programming schemes, known as polymorphism and inheritance, to implement inter-module communications. The derivation of simulation module classes from the abstract base class *Generic Memory Module* and overloading the virtual functions *request*, *respond*, and *cancel* in each class allow CaPSim to select the appropriate function at run-time. As mentioned earlier, the simulation set contains pointers to dynamically allocated simulation modules. Each simulation module communicates with its master and slave by using their pointers from the simulation set. Although these pointers are all of the same type (*Generic Memory Module*), function calls through different pointers will invoke functions in different class instances, depending on the run-time resolution of the virtual functions. Therefore, a simulation module does not have to know the type of its master and slave to make a request or response.

Figure 39 shows the module instances created during the simulation phase, based on the example configuration file of Figure 37. The module identification numbers represent corresponding indices into the simulation set. Although the *Event Queue* is embedded in the *CPU* class, it is depicted separately to demonstrate its functional relationship with the simulation modules.

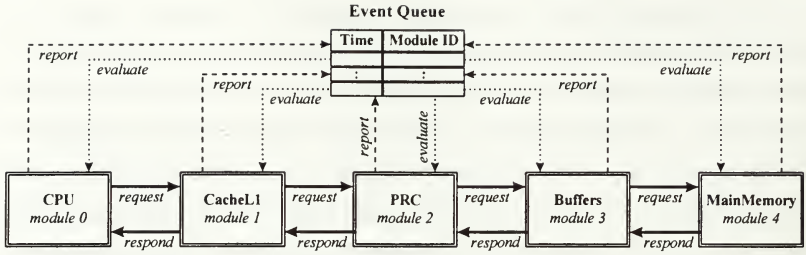

Figure 39. Module Instances Created During the Simulation Phase

The *Generic Memory Module* provides another virtual function, *evaluate()*, to simulate internal events reported by a simulation module. Each simulation module reports its time-dependent internal events to the *Event Queue* by passing its module ID and the target time of the event. The *Event Queue* sorts these events with respect to their target times and retires them by calling the *evaluate* function of their corresponding simulation modules. This approach gives simulation modules a chance to switch their states before the system clock is advanced to the target time of the next event in the queue. It also improves the simulation efficiency because only those modules that actually need a change in their states are evaluated.

The actions taken by the *evaluate* function depend on the state machine of a particular simulation module. Table 25 lists possible states implemented in software for each simulation module.

The *Main Memory* class has the simplest state machine with only three states. It switches from the *Idle* to the *Access* state upon receiving a request from the downstream memory levels. In order to switch to the *Transfer* state after the memory access time elapses, it reports its target time (Current Time + Memory Access Time) to the *Event Queue*. When its *evaluate* function is called, it switches to the *Transfer* state and reports the finish time of the transfer to switch back to the *Idle* state. The access and transfer times are both assumed to be the same for both read and write transactions.

CPU	Cache	PRC	Buffer Module	Main Memory
Boot	Idle	Idle	Idle	Idle
Running	Read Access	Read Access	Read Access	Access
Read Stall	Write Access	Write Access	Write Access	Transfer
Write Stall	Read Hit	Read Hit	Read Transfer	
	Read Miss	Read Miss	Write Transfer	
	Write Hit	Write Hit		
	Write Miss	Write Miss		
	Read Transfer	Read Transfer		
	Write Transfer			
	Update			
	Write Back			
	Write Allocate			

Table 25. Simulation Module States

The total transfer time is determined by the user-defined *Transfer Size* parameter that represents the bus-width between two modules. The transfer size is known only by the slave module. By using the requested size, the transfer size, and the time required to transfer a single word (*Transfer Time*), the slave module calculates the total time it will take to complete a transfer and reports this time to the master module through the *respond* function. It is the responsibility of the salve to recalculate the transfer time and respond again in case the master module is busy.

The *CPU* class is initialized in the *Boot* state at the beginning of a simulation. It switches to the *Read Stall* or *Write Stall* state each time it makes a read or write request, respectively. When it receives a response from its slave, it assumes the *Running* state until the next read or write stall.

The *Cache* class contains the largest set of states in order to simulate as many configurations as possible. However, it uses only a subset of its states during a simulation, depending on the cache policies defined by the user. Figure 40 shows the overall state diagram of the *Cache* class. The transitions between states are affected by the write policy, the write miss policy, the use of a block buffer, and the fetch policy of the cache.

The block buffer is used to speed up block updates in the cache upon a read miss [Ref. 5, p. 82]. It allows the CPU to continue execution instead of holding in a wait state until the entire block is transferred from the memory. Block buffers are generally used in conjunction with a streaming fetch policy so that the missed word is fetched from the

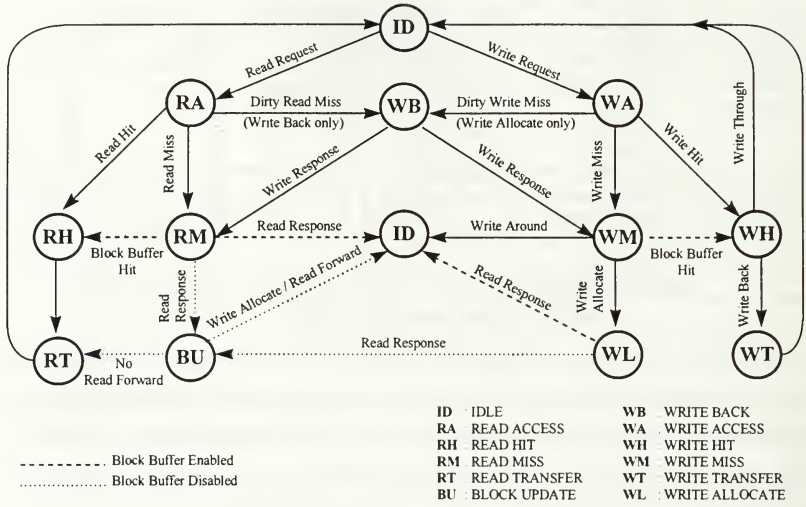

Figure 40. Cache State Machine

memory first and forwarded to the CPU at the same time it is being written into the block buffer [Ref. 5, p. 83]. The cache can continue to service new CPU requests while the rest of the block is being transferred into the block buffer from an upstream memory level. This fetch scheme is referred to as "*desired word first*" fetch , *wrapping* fetch, or *read forwarding* [Ref. 2, p. 90]. CaPSim provides the user with three input parameters to specify the behavior of the cache during a block update: *Enable Block Buffer*, *Read Forward*, and *Wrapping Fetch Policy*. The wrapping fetch policy is specified as either *Wrap Up* or *Wrap Down*, depending on the desired fetch direction of the remaining bytes in the block buffer. It should be noted that using a wrapping fetch policy without a block buffer will not provide any advantage because of the wait states imposed on the CPU.

The transitions that are illustrated with solid lines in Figure 40 are not affected by the choice of enabling or disabling the block buffer. The dashed lines represent transitions with a block buffer and the dotted lines represent transitions without a block buffer. The *Idle* state (ID) is denoted with two separate nodes for clarity. There are four different factors

that determine a transition between two states: external events (requests and responses), cache policies (write policy, write miss policy, and read forwarding), the access status from the *Cache Logic* class (hit, miss, dirty miss), and the expiration of a user-defined timing parameter.

There are six timing parameters in Table 22 that determine the number of cycles spent in the corresponding cache states: *Read Access Time* (RA), *Write Access Time* (WA), *Read Hit Time* (RH), *Read Miss Time* (RM), *Write Hit Time* (WH), and *Write Miss Time* (WM). The duration of the *Read Transfer* state and *Write Transfer* state is determined by the transfer size between the cache and its master module, as explained for the *Main Memory* class.

As mentioned earlier, the *Cache* class only implements the state machine and leaves the low-level details to the *Cache Logic* class. The communication between these two classes is performed with a number of function calls through the *Generic Logic* class. The functions *read* and *write* are of major importance to determine the status of the cache access. The *Cache* class passes an incoming request directly to the *Cache Logic* class and makes a transition decision depending on the values returned by the *read* and *write* functions. Both of these functions return three status messages after searching for the request in the cache: hit, miss, and dirty miss. A dirty miss is either caused by a read miss or a write miss (with write allocate policy) in a cache block that contains dirty data [Ref. 5, p. 64].

The state machine of the *PRC* is a subset of the *Cache* state machine because the *PRC* does not use write back and write allocate policies. In addition, the state transitions are simpler because the block buffer in the *PRC* class is always enabled by default. The access status is determined by the *PRC Logic* or *iPRC Logic* class, depending on the prediction algorithm being used. These classes have the same interface as the *Cache Logic* class.

The block buffer in both *Cache* and *PRC* class has its own state machine made of four states: *Idle*, *Transfer*, *Ready*, and *Block Update*. The *Transfer* state indicates that a block is being transferred into the block buffer from the upstream memory levels. After the

transfer is completed, the block buffer switches to the *Block Update* state, provided that the cache (or the PRC) is not busy. If it is busy, then the block buffer waits in the *Ready* state until the cache is available for update. Therefore, the cache can switch from the *Read Miss* state to the *Idle* state as soon as the block buffer transfer starts.

3. Evaluation

The evaluation phase involves the calculation of temporal and statistical results based on the information collected during a simulation. CaPSim creates a separate ASCII file in the output directory for each module instance used in the simulation. It also saves the module contents in binary files so that they can be used cumulatively in the following simulations. The filenames are formed by appending either "*_dump*" or "*_save*" to module names defined in the configuration file. All files are given the extension of the last trace file used in the simulation. An example output file is shown in Appendix D through Appendix H for each of the five simulation modules.

D. INTEGRATED DEBUGGER

CaPSim can be run in the debug mode by giving the argument "*-debug*" at the command-line following the configuration filename. The integrated debugger is designed to assist the programmer in solving potential software problems associated with CaPSim's source code. It provides useful run-time information that helps the programmer understand the behavior of CaPSim with different configurations.

CaPSim cannot be interrupted during a normal simulation run until the end of the evaluation phase. However, the integrated debugger can run a simulation in steps by interrupting the main event loop during the simulation phase. It does not affect the operation of the configuration and evaluation phases. The debug mode is tested by using a single conditional statement within the main event loop.

The programmer can interact with the debugger through a user-interface that is embedded in the *CPU* class. There are two types of debug commands: global and local. Global commands are handled by the *CPU* class and the local commands are passed to the individual module instances. The command-line prompt provided by the debugger contains

the name of the current simulation module that is being debugged. The local commands are recognized only when their corresponding modules are being prompted. The programmer can change the command-line prompt to another module by entering either the name or the identification number of the desired module. On the other hand, global commands are always recognized by the debugger, regardless of the command-line prompt. Table 26 lists the global debug commands implemented in the current version.

Command	Shortcut	Description
queue	eq	Display the contents of the event queue
list	lm	Display the module list
time	t	Display the current simulation time
state	st	Display the current state of the module being prompted
states	ss	Display the current states of all modules in the simulation set
hierarchy	mh	Display the memory hierarchy
next	n	Change command-line prompt to the next module in the simulation set
param	pp	Display the user-defined input parameters of the current module
stats	ps	Display the statistics collected so far for the current module
step	+	Step until the next target time in the event queue (single step)
run *time*	++ *time*	Run until the specified simulation time
file *ext*	f+ *ext*	Run until the end of the trace file with the specified file extension
module name		Change command-line prompt to the specified module
module ID		Change command-line prompt to the specified module
help	?	Display on-line help
exit		Exit CaPSim

Table 26. Global Debug Commands

Each simulation module defines its own local debug commands in a virtual function called *debug*. If the *CPU* class cannot recognize a command, it calls the *debug* function of the current simulation module with a string argument. The simulation module processes the command and returns the result to the debugger. Table 27 shows the local commands and the simulation modules that implement them.

Command	Shortcut	Description	Simulation Modules
trace	ti	Display trace file information	CPU
inreq	ir	Display the input request	Cache, PRC, Buffer Module, Main Memory
outreq	ro	Display the output request	CPU, Cache, PRC, Buffer Module
inres	ri	Display the input response	CPU, Cache, PRC, Buffer Module
outres	or	Display the output response	Cache, PRC, Buffer Module, Main Memory
pending	pr	Display the pending request	Cache, PRC, Buffer Module, Main Memory
decoder	ad	Display the address decoder	Cache, PRC
target	tt	Display the target time	Cache, PRC
buffer	bt	Display the buffer time	Cache, PRC
ractive	ar	Display the active read	Buffer Module
wactive	aw	Display the active write	Buffer Module
read	rb	Display the read buffer	Buffer Module
write	wb	Display the write buffer	Buffer Module

Table 27. Local Debug Commands

VI. SIMULATION RESULTS AND ANALYSIS

A. ASSUMPTIONS

CaPSim uses an analytical simulation model in order to determine the performance of a given memory hierarchy. It is backward compatible with the SACS2 simulator used by Miller [Ref. 8, p. 19] and supports all the assumptions built into SACS2. However, there are only a few hard-coded assumptions in CaPSim. Most of the assumptions are made in the configuration phase by using the simulation language provided by CaPSim.

Fouts and Billingsley have discussed that a PRC should be used to predict the read miss patterns in the first-level data cache without interfering with the operation of the instruction cache [Ref. 6, p. 110]. Although the current version of CaPSim does not simulate an instruction cache, it accounts for the memory cycles consumed by instruction references in evaluating the performance for data references.

The simulations in this thesis research are primarily aimed at comparing the performance of the new PRC design with the performance of the original design and two-level cache hierarchies. Miller has already shown that varying the PRC size, the degree of associativity, and policy parameters does not provide a significant improvement in the performance [Ref. 8]. Therefore, the simulations are performed in order to observe the impact of the new prediction algorithm on the performance, rather than the impact of the organizational PRC parameters and policies.

B. CONSTANT PARAMETERS

All user-defined parameters except the PRC prediction algorithm, the PRC size, and the degree of associativity are set to constant values in all simulations. The following subsections summarize these constant parameters and their influence on the memory hierarchy. The only parameter that is shared among all modules in the simulation set is the word size. It is defined within the simulation block of the configuration file and is set to 4 bytes in all simulations.

1. First-level Cache Parameters

Address traces used in the simulations are collected from a SPARCstation 1+ running SUN OS Release 4.1.2 [Ref. 14]. Therefore, the first-level cache parameters are set according to those implemented in the SPARCstation 1+ and are listed in Table 28.

Parameter Name	Value	Parameter Name	Value
Cache Size	65536	Access Time	1
Block Size	16	Read Hit Time	0
Sub-block Size	4	Read Miss Time	0
Fetch Size	16	Write Hit Time	0
Transfer Size	4	Write Miss Time	0
Associativity	1	Block Buffer Transfer Time	1
Write Policy	Write Through	Enable Block Buffer	Yes
Write Miss Policy	Write Around	Search Block Buffer	Yes
Wrapping Fetch Policy	Wrap Up	Read Forward	Yes

Table 28. Constant First-Level Cache Parameters

The first-level cache is a 64-Kbyte direct-mapped cache with a block size of 16 bytes and a sub-block size of four bytes. The sub-block size is used to implement a sectored cache in which a separate valid bit is used for each of the 4-byte sectors in a 16-byte cache block [Ref. 2, p. 11]. The fetch size determines the number of bytes that must be fetched from the upstream memory levels after a read miss occurs in the cache. CaPSim is capable of simulating multi-block fetches for a single read miss. However, in this particular case, the fetch size is set to the block size (16 bytes) in order to simulate single-block fetches.

The transfer size is used to specify the bus width in number of bytes between the CPU and the first-level cache. As mentioned in Chapter V, the transfer sizes are determined by the slave modules. Therefore, the bus width between the first-level cache and the PRC is specified by the transfer size defined in the PRC.

Since the cache has a direct-mapped organization with a single degree of associativity, no replacement policy needs to be defined in the configuration file. On the other hand, the write policy is set to write through and the write miss policy is set to write around. The wrapping fetch policy defines the direction of fetches from the upper memory levels during a block update.

The access time parameter sets both the read and write access times to one cycle. CaPSim also allows the user to define these access times separately by using the parameter names "read access time" and "write access time." All hit and miss times are set to zero cycles for the first-level cache. Thus, at the end of the 1-cycle cache access, hit and miss actions for both read and write transactions can be taken without any additional cycle-time cost.

The block buffer in the first-level cache is enabled and the time required for the block buffer to update a cache block is set to one cycle. The read forwarding is also enabled to fetch the requested word first and allow the CPU to continue its execution during the block-buffer transfer.

2. PRC Parameters

The constant parameters used in the PRC are shown in Table 29. The PRC block size is set to the same value as the block size of the first-level cache. Both the sub-block size and the fetch size are set to the block size (16 bytes) by CaPSim, if they are not specified by the user. Since the block size and the sub-block size are equal, the PRC contains a single valid bit for each block. The transfer size is defined as 16 bytes in order to simulate single-cycle transfers between the PRC and the first-level cache.

Parameter Name	Value
Block Size	16
Transfer Size	16
Replacement Policy	LRU
Write Policy	Write Through
Access Time	0
Read Hit Time	1
Read Miss Time	1
Block Buffer Transfer Time	1
Bypass Write Allocates	Yes
Minimum read size in buffer	12
Maximum read slips in buffer	2

Table 29. Constant PRC Parameters

The replacement policy is selected as LRU in all simulations. The write policy is specified as write through and the write miss policy is implicitly set to write around by CaPSim.

The access time of the PRC is always defined relative to the access time of the first-level cache. An access time of zero indicates that the cache and the PRC start their accesses simultaneously. However, if it is desired to start after the first-level cache accesses are completed, then a non-zero access time must be specified in the PRC.

The read hit time is used to simulate the cycle-time cost of forwarding the predicted data from the PRC to the cache when a first-level cache read miss hits in the PRC. However, it is the responsibility of the user to define the transfer size and the read hit time consistently. For example, if the bus-width between the PRC and the cache were four bytes, then a read hit time of one cycle would not be realistic. The read hit time is also used by the PRC to make a prefetch request from the upper memory levels after forwarding the data to the cache.

The read miss time is used by the PRC to calculate the predicted miss address and make a prefetch request when the read request misses both in the first-level cache and the PRC.

The "minimum read size in buffer" specifies the minimum number of bytes that must be transferred into the PRC in order to allow a PRC read transaction to continue. Normally, the PRC transactions are interrupted by higher-priority cache transactions in the read buffer. However, this parameter indicates that a PRC transaction should continue if at least 12 bytes out of 16 bytes are already transferred into the PRC. The interaction between the PRC and cache transactions is fully described by Miller in Ref. 8.

The "maximum read slips in buffer" specifies the number of times that a PRC request is allowed to slip from the top of the read buffer. Miller has implemented this parameter in SACS2 with the name *DropPRCOnSecondTry*. In CaPSim, a value of two indicates that a PRC request must be dropped out of the read buffer after two slips. A value of zero disables the drops caused by multiple slips.

3. Transaction Priorities

The transaction priorities are implicitly set by CaPSim if the user does not define specific priority values. CaPSim uses a priority coefficient to reserve a priority interval for the simulation modules by multiplying their module IDs with this coefficient. For example, if the priority coefficient is 10 and the module IDs for the first-level cache and the PRC are 1 and 2, then their priority intervals start from 10 and 20, respectively. If the default read priority is 1, then the first-level cache reads are given a priority of 11 (10+1) and the PRC reads are given a priority of 21 (20+1). The lower values indicate higher priorities. The transaction priorities are used by the buffer module in selecting requests from the read and write buffers before making a request to the main memory.

4. Buffer Module Parameters

The constant buffer module parameters are shown in Table 30. The sizes of read and write buffers are set to 8 and 4, respectively. The write buffer block size specifies the number of bytes that can be stored into a single buffer line. This value is used by the buffer module to merge adjacent write requests into a single write request.

The transaction priorities are enforced in the buffer module in order to process the requests according to their priorities. The buffer module is also allowed to remove the duplicate read and write requests from the buffers. The "search write buffer" parameter is used to determine whether the incoming read requests hit in the write buffer. The "search read buffer" parameter is used to update the read buffer in case a write request hits in the buffer.

Parameter Name	Value
Read Buffer Size	8
Write Buffer Size	4
Write Buffer Block Size	16
Enforce Priorities	Yes
Remove Read Duplicates	Yes
Remove Write Duplicates	Yes
Search Read Buffer	Yes
Search Write Buffer	Yes

Table 30. Constant Buffer Module Parameters

5. Main Memory Parameters

The constant parameters of the main memory are shown in Table 31. The access time is set to 5 cycles and the transfer time is set to one cycle. The main memory is assumed to service the first word of the request at the end of a 5-cycle memory access. Then, it services the remaining words one cycle at a time. The transfer size determines the bus width between the main memory and the downstream memory levels.

Parameter Name	Value
Access Time	5
Transfer Time	1
Transfer Size	4

Table 31. Constant Main Memory Parameters

C. SIMULATION RESULTS

The simulations are performed by using address traces containing the Kenbus20 and Kenbus80 benchmarks. Table 32 shows the baseline performance results of a memory hierarchy with only a first-level cache. These results will be used to calculate the speedup obtained by placing a PRC or a second-level cache between the first-level cache and the main memory.

(L1 Only) Benchmark	Average Read Access Time	Cache Read Hit Rate	Average Write Access Time	Cache Write Hit Rate
Kenbus20	1.51300573	89.94 %	1.00000000	64.32 %
Kenbus80	1.72102642	86.44 %	1.00000000	63.90 %

Table 32. Baseline Performance with only a First-Level Cache

As mentioned in Chapter IV, the only difference between the Kenbus20 and Kenbus80 benchmarks is the number of users running the same benchmark concurrently. The increase in the number of supervisor references and context switches in Kenbus80 degrades the read hit rate of the first-level cache and the average read access time of the system.

1. Second-Level Cache Simulations

In order to compare the PRC and the second-level cache performance, a number of second-level cache simulations are performed prior to the PRC simulations. The second-

level cache sizes are varied from 64 Kbytes to 512 Kbytes by doubling the size in each simulation. Although a second-level cache with the same size as the first-level cache is not expected to improve the performance, the 64-Kbyte size is included in simulations to demonstrate the general trend in the two-level cache hierarchies.

All second-level cache simulations use a 4-way set-associative organization with an LRU replacement policy. The write policy is selected as write through and the write miss policy is selected as write around. The access time is defined as one clock cycle in order to simulate on-chip second-level caches.

The results obtained from these simulations are shown in Table 33 in terms of the average read access time and the corresponding speedup over the baseline first-level cache performance. As expected, a 64-Kbyte second-level cache provides only minuscule improvement in the performance. The speedup increases as the second-level cache size increases.

Second-Level Cache Size	Kenbus20		Kenbus80	
	Average Read Access Time	Speedup	Average Read Access Time	Speedup
64 Kbytes	1.505376	0.50 %	1.654211	3.88 %
128 Kbytes	1.413992	6.54 %	1.485237	13.70 %
256 Kbytes	1.30822	13.54 %	1.318546	23.39 %
512 Kbytes	1.210122	20.02 %	1.220529	29.08 %

Table 33. Second-Level Cache Performance

2. 4-Way Set-Associative PRC Simulations

The 4-way set-associative PRC simulations are performed by using two different prediction algorithms. The PRC size is varied from 256 bytes to 512 Kbytes by doubling the size in each simulation. Although large PRC sizes are not of major interest, they are included to observe the general trend of the PRC performance. Tables 34 and 35 summarize the results obtained from Kenbus20 and Kenbus80 simulations, respectively. The original PRC design is denoted with *d-PRC* (data-address-tagged PRC) and the new PRC design is denoted with *i-PRC* (instruction-address-tagged PRC).

PRC Size (bytes)	d-PRC			i-PRC		
	Average Read Access Time	Speedup	PRC Read Hit Rate	Average Read Access Time	Speedup	PRC Read Hit Rate
256	1.36965531	9.47%	15.26%	1.39045644	8.10%	26.84%
512	1.36879874	9.53%	15.88%	1.38972211	8.15%	27.10%
1K	1.36710871	9.64%	17.66%	1.32430685	12.47%	36.92%
2K	1.36659884	9.68%	18.70%	1.32035351	12.73%	37.57%
4K	1.36388659	9.86%	19.38%	1.31997263	12.76%	37.73%
8K	1.3607111	10.07%	20.09%	1.32005227	12.75%	37.81%
16K	1.35767436	10.27%	20.89%	1.31995714	12.76%	37.89%
32K	1.35272896	10.59%	22.71%	1.3198719	12.76%	37.96%
64K	1.3453083	11.08%	23.61%	1.31989872	12.76%	37.98%
128K	1.3299576	12.10%	33.28%	1.31987596	12.76%	37.99%
256K	1.30965281	13.44%	40.27%	1.31986213	12.77%	37.99%
512K	1.28768337	14.89%	43.52%	1.31986821	12.77%	37.99%

Table 34. 4-Way Set-Associative PRC Performance for Kenbus20

PRC Size (bytes)	d-PRC			i-PRC		
	Average Read Access Time	Speedup	PRC Read Hit Rate	Average Read Access Time	Speedup	PRC Read Hit Rate
256	1.53792291	10.64%	7.60%	1.54227054	10.39%	25.71%
512	1.53762291	10.66%	7.96%	1.54166102	10.42%	25.93%
1K	1.53732291	10.67%	10.30%	1.3989538	18.71%	42.30%
2K	1.53704	10.69%	11.51%	1.39765811	18.79%	42.61%
4K	1.53604043	10.75%	11.93%	1.39823127	18.76%	42.70%
8K	1.53347158	10.90%	12.36%	1.3985312	18.74%	42.78%
16K	1.52970231	11.12%	13.03%	1.3983264	18.75%	42.90%
32K	1.52292323	11.51%	14.03%	1.39799845	18.77%	43.03%
64K	1.51063859	12.22%	25.93%	1.39786088	18.78%	43.07%
128K	1.48218095	13.88%	31.28%	1.3978399	18.78%	43.08%
256K	1.436234	16.55%	35.12%	1.3978554	18.78%	43.07%
512K	1.36879456	20.47%	41.15%	1.39785576	18.78%	43.07%

Table 35. 4-Way Set-Associative PRC Performance for Kenbus80

The changes in the average read access time and the speedup are plotted in Figures 41, 42, 43, and 44 as a function of the PRC size. These plots include both the i-PRC and the d-PRC results, as well as the results obtained from second-level cache simulations.

For the PRC sizes of 256 bytes and 512 bytes, the d-PRC performs slightly better than the i-PRC because the load instructions systematically override each other in the set-associative i-PRC organizations. In a 256-byte 4-way set-associative i-PRC, there are only

four sets and the low-order two bits of the instruction address are used as an index. Therefore, most of the instructions map into the same block and prevents the i-PRC from retaining the miss patterns of the instructions.

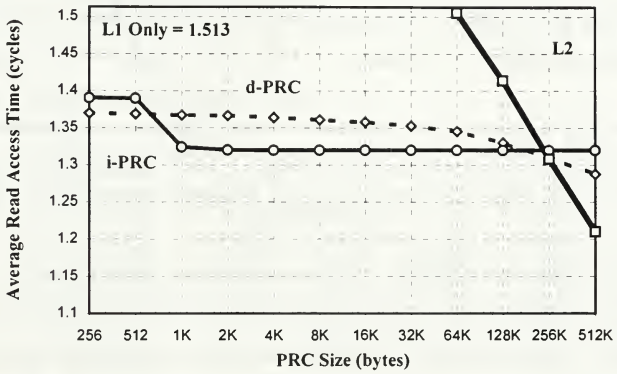

Figure 41. Average Access Time vs. PRC Size for Kenbus20 (4-Way Set-Associative)

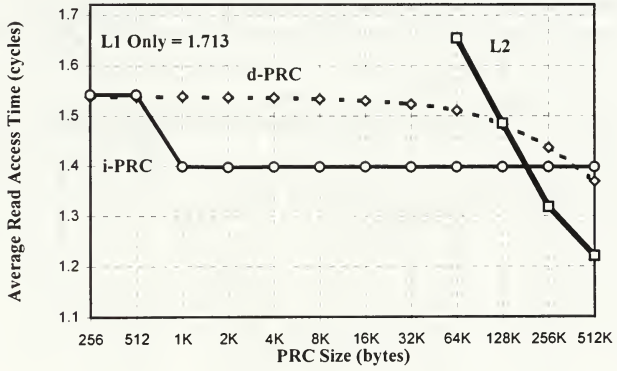

Figure 42. Average Access Time vs. PRC Size for Kenbus80 (4-Way Set-Associative)

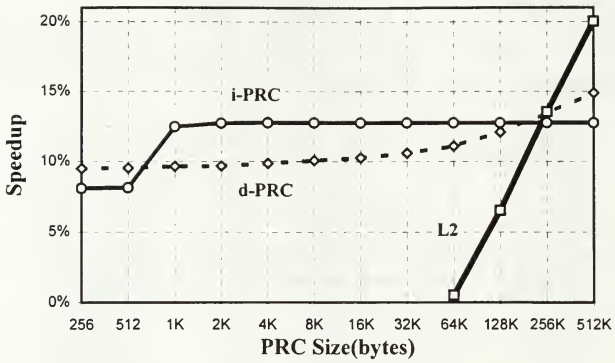

Figure 43. Speedup vs. PRC Size for Kenbus20 (4-Way Set-Associative)

The i-PRC outperforms the d-PRC for sizes above 1 Kbyte and below 256 Kbyte. The improvement in the average access time for Kenbus80 benchmark is more than that for Kenbus20.

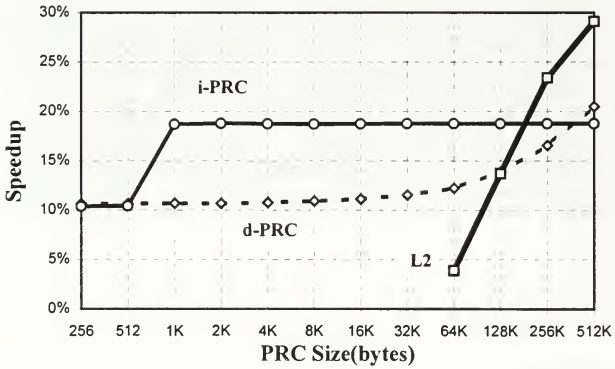

Figure 44. Speedup vs. PRC Size for Kenbus80 (4-Way Set-Associative)

As the PRC size increases, the d-PRC behaves like a second-level cache and provides more speedup than the i-PRC. However, for these large PRC sizes, the

second-level cache performs better than both of the PRC designs in this region. The speedup obtained from the i-PRC is almost constant for sizes above 1 Kbyte.

3. Fully-Associative PRC Simulations

The PRC simulations are repeated for fully-associative organizations and the results are summarized in Tables 36 and 37. The average read access time and the speedup are plotted as a function of the PRC size in Figures 45, 46, 47, and 48.

PRC Size (bytes)	d-PRC			i-PRC		
	Average Read Access Time	Speedup	PRC Read Hit Rate	Average Read Access Time	Speedup	PRC Read Hit Rate
256	1.36902398	9.52%	15.47%	1.32261944	12.58%	37.49%
512	1.3688836	9.53%	17.94%	1.31724226	12.94%	38.40%
1K	1.36866736	9.54%	18.37%	1.31350088	13.19%	39.15%
2K	1.36547351	9.75%	18.88%	1.31168818	13.31%	39.57%
4K	1.36352956	9.88%	19.36%	1.31065762	13.37%	39.94%
8K	1.36181247	9.99%	19.84%	1.30916977	13.47%	40.40%
16K	1.35902596	10.18%	20.40%	1.30642736	13.65%	41.11%
32K	1.35664368	10.33%	21.00%	1.3019464	13.95%	42.30%
64K	1.35070193	10.73%	22.23%	1.29702485	14.27%	43.54%
128K	1.34257758	11.26%	24.30%	1.29636526	14.32%	43.70%
256K	1.33130932	12.01%	28.10%	1.29636526	14.32%	43.70%
512K	1.32281983	12.57%	32.88%	1.29636526	14.32%	43.70%

Table 36. Fully-Associative PRC Performance for Kenbus20

PRC Size (bytes)	d-PRC			i-PRC		
	Average Read Access Time	Speedup	PRC Read Hit Rate	Average Read Access Time	Speedup	PRC Read Hit Rate
256	1.54600349	10.17%	7.67%	1.3971287	18.82%	42.61%
512	1.54003417	10.52%	11.17%	1.39340544	19.04%	43.18%
1K	1.53887475	10.58%	11.36%	1.39089119	19.18%	43.68%
2K	1.53746867	10.67%	11.61%	1.38972223	19.25%	44.05%
4K	1.5362767	10.73%	11.86%	1.3892355	19.28%	44.29%
8K	1.53415942	10.86%	12.25%	1.38732874	19.39%	44.80%
16K	1.53100729	11.04%	12.68%	1.38307083	19.64%	45.63%
32K	1.52639592	11.31%	15.77%	1.37514114	20.10%	47.09%
64K	1.51809323	11.79%	14.49%	1.37351847	20.19%	47.39%
128K	1.49989676	12.85%	17.71%	1.37351847	20.19%	47.39%
256K	1.46887887	14.65%	25.66%	1.37351847	20.19%	47.39%
512K	1.44398022	16.10%	30.38%	1.37351847	20.19%	47.39%

Table 37. Fully-Associative PRC Performance for Kenbus80

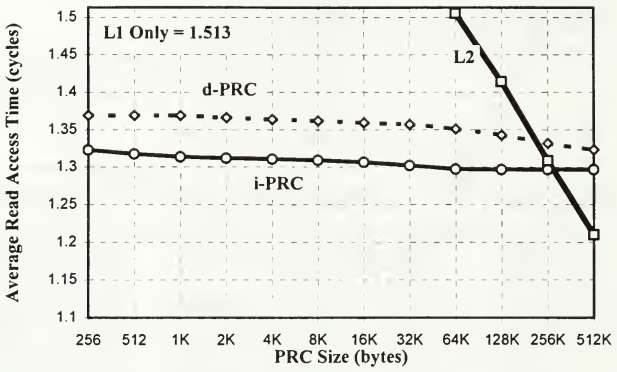

Figure 45. Average Access Time vs. PRC Size for Kenbus20 (Fully-Associative)

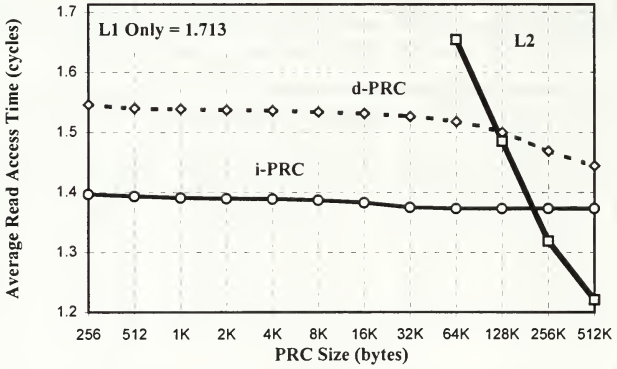

Figure 46. Average Access Time vs. PRC Size for Kenbus80 (Fully-Associative)

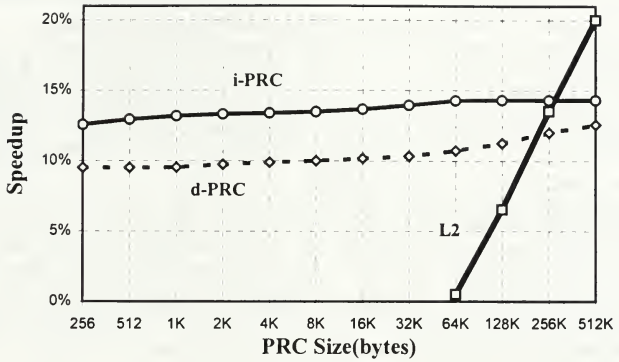

Figure 47. Speedup vs. PRC Size for Kenbus20 (Fully-Associative)

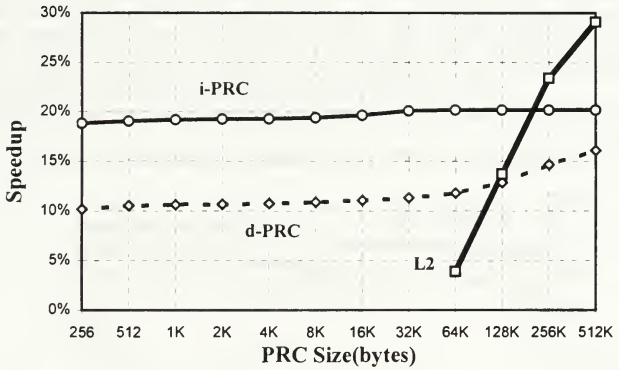

Figure 48. Speedup vs. PRC Size for Kenbus80 (Fully-Associative)

In fully-associative PRC simulations, the i-PRC outperforms the d-PRC for all sizes. This is attributed to the fact that the i-PRC can now allocate the instruction addresses as they are needed, without any restrictions in the mapping function. The d-PRC still converges to the behavior of a second-level cache as the PRC size increases.

Both Figure 45 and 46 reveal that the i-PRC saturates above a certain PRC size. Increasing the size does not provide any additional improvement in the performance, once

all instructions in the working set can be accommodated in the i-PRC. The saturation starts at the size of 128 Kbytes for Kenbus20 and at the size of 64 Kbytes for Kenbus80.

The speedup of the i-PRC over the d-PRC ranges from 2% to 3.4% for Kenbus20, and from 4.9% to 9.6% for Kenbus80. On the other hand, the speedup of a 1-Kbyte fully-associative i-PRC can only be balanced with a 128-Kbyte or 256-Kbyte on-chip second-level cache, depending on the particular benchmark.

As the operating system code and context switches degrade the first-level cache performance in the Kenbus80 benchmark, the i-PRC performs better. Since more instructions miss in the first-level data cache in Kenbus80, the i-PRC operates on a larger working set and uses more run-time information to make predictions.

Figure 49 illustrates the difference in the hit rates of the two PRC designs for both Kenbus20 and Kenbus80. The upper and lower parts of the graph represent the hit rates for the i-PRC and the d-PRC, respectively. The d-PRC hit rates tend to decrease with the increased number of users and operating system activity. However, under the same circumstances, the i-PRC hit rates increase when the benchmark is changed from Kenbus20 to Kenbus80. These results are consistent with the speedup trends discussed earlier.

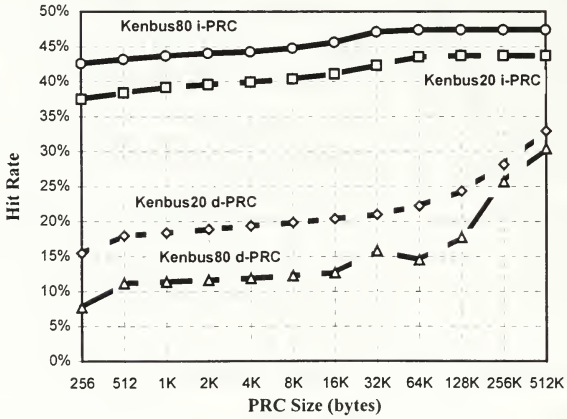

Figure 49. Hit Rate vs. PRC Size for Fully-Associative Organizations

D. COST/PERFORMANCE

The performance results obtained from the i-PRC simulations are combined with the hardware cost estimates from Chapter III in order to determine the optimum cost/performance choice for fully-associative organizations. Figure 50 shows the variation in the cost/performance as a function of the PRC size.

The cost/performance ratio is calculated relative to the original PRC design, i.e., the increase in the transistor count is divided by the increase in the speedup. The left and right vertical axes represent the cost/performance scale for Kenbus20 and Kenbus80, respectively. The optimum cost/performance is given by the minimum points of the curves. The most cost-effective sizes are 1 Kbyte for Kenbus20 and 256 bytes for Kenbus80. Both curves also have a local minimum at 32 Kbytes.

These small sizes provide a speedup of more than 13% for Kenbus20 and 20% for Kenbus80, over the baseline performance of the first-level cache. In both cases, only a 256-Kbyte on-chip second-level cache can perform better than the i-PRC.

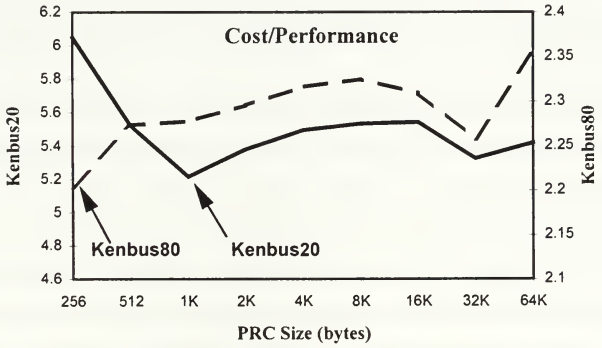

Figure 50. Variation in Cost/Performance as a Function of PRC Size

VII. CONCLUSION

A. SUMMARY

This thesis research has developed a new prediction algorithm for the PRC that can track multiple miss patterns in the first-level data cache with respect to the addresses of the instructions that generate the data read. The simulation results prove that a PRC using the new prediction algorithm can provide a significant improvement to the memory hierarchy. It improves the performance of the original PRC under multitasking workloads and removes the sensitivity to poor locality of reference. The improvement becomes more significant as more operating system calls, context switches, and branches pollute the first-level data cache.

The direct-mapped, set-associative, and fully-associative design alternatives show that the new algorithm does not place any limitations on the PRC organization. The migration from the data-address-tagged PRC to the instruction-address-tagged PRC can be accomplished with an additional SRAM containing instruction address tags and with some minor changes to the PRC controller logic. The architectural support for the new design can be provided by a dedicated instruction-address bus between the CPU and the PRC. The new prediction algorithm does not affect the interface between the CPU and the first-level data cache.

Feasibility studies for a VLSI implementation have provided an understanding of the tradeoffs between the cost and performance. The estimated transistor counts also give an insight into the power dissipation problem, which is especially important for high-performance embedded systems. The additional transistor cost of the new design has proven to be tolerable for sizes less than 32 Kbytes. These sizes are of major interest because PRC sizes above 128 Kbytes are not as advantageous as a second-level cache of the same size.

The conversion of address traces has provided the means for a simulation environment for the new algorithm. Although the conversion program is designed specifically for the SPARC architecture, it describes the basic algorithm for extracting the

required information from the address traces. The program can easily be modified for the conversion of other traces captured from different platforms.

The trace conversion program also provides statistical information about the address traces being used in simulations. This information, together with the simulation results, is very helpful in understanding the behavior of the memory hierarchy under different workloads.

The simulator developed in this thesis is a flexible and efficient simulation tool that can simulate a wide range of configurations in the memory hierarchy. It can be used for both the original and the new PRC simulations, as well as simulations of multi-level cache hierarchies. The capabilities of the simulator can easily be extended due to the object-oriented programming techniques employed in the source code.

B. RECOMMENDATIONS

The simulation results obtained in this thesis research are consistent with those of the previous PRC studies. The new PRC design outperforms the original design and provides a speedup of 13% to 20% over the baseline performance of a single first-level data cache. This performance improvement can only be obtained by using a 256-Kbyte second level cache, which is approximately 1,000 times larger than the PRC. These results yield great promise for the PRC to replace larger, second-level cache memories and save cost and power consumption.

The cost/performance analysis of the new PRC design shows that a small PRC provides almost the same performance improvement as a large one. Therefore, a 256-byte to 2-Kbyte, fully-associative PRC is recommended as the optimum organization for future implementations.

The performance of the new PRC design needs to be investigated further by converting longer address traces and simulating a larger set of design alternatives. In addition, more aggressive simulations can be performed by using the PRC in place of the first-level data cache. The advent of on-chip, second-level cache memories qualifies the PRC as a potential candidate for the third memory level between the second cache and the

main memory. All of these alternative configurations can be simulated by using CaPSim, without any need to modify the source code.

The new PRC design will allow microprocessor systems to operate at high speeds without the need for a second-level cache. This will decrease the amount of required hardware, the power consumption, and the size and weight of high-performance microprocessor systems. Thus, the PRC is of major importance to embedded systems in space-based, weapons-based and portable/mobile computing applications.

APPENDIX A.

TRACE OUTPUT FILE CREATED BY BATE

```
0008530 : fd801e71 00171803 0007 1 0 1 001 1 0 : [READ]/1
0008531 : f80eed50 ea2e2039 0007 1 0 0 004 1 0 : [STB]
0008532 : f80eed54 2d3e05a9 0000 1 0 0 004 1 0 : [SETHI]
0008533 : f80eed58 ec05a3dc 0000 1 0 0 004 1 0 : [LD]
0008534 : fd801e71 00170000 0001 1 1 1 001 1 0 : [WRITE]/1
0008535 : f80eed5c 80a5a001 0000 1 0 0 004 1 0 : [SUBcc]
0008536 : f80eed60 2480000f 0000 1 0 0 004 1 0 : [BLE]
0008537 : f816a7dc 00000000 0000 1 0 1 004 1 0 : [READ]/4
0008538 : f80eed64 808f6080 0001 1 0 0 004 1 0 : [ANDcc]
0008539 : f80eed9c 22800004 0000 1 0 0 004 1 0 : [BE]
0008540 : f80eeda0 808f6040 0000 1 0 0 004 1 0 : [ANDcc]
0008541 : f80eedac 22800008 0000 1 0 0 004 1 0 : [BE]
0008542 : f80eedb0 808f6003 0000 1 0 0 004 1 0 : [ANDcc]
0008543 : f80eedcc 22800004 0000 1 0 0 004 1 0 : [BE]
0008544 : f80eedd0 808f6004 0000 1 0 0 004 1 0 : [ANDcc]
0008545 : f80eeddc 22800012 0000 1 0 0 004 1 0 : [BE]
0008546 : f80eede0 f40e2037 0000 1 0 0 004 1 0 : [LDUB]
0008547 : f80eee24 808ea0e0 0000 1 0 0 004 1 0 : [ANDcc]
0008548 : f80eee28 22800004 0000 1 0 0 004 1 0 : [BE]
0008549 : fd801e6f 0004a020 0000 1 0 1 001 1 0 : [READ]/1
0008550 : f80eee2c f60e2037 0001 1 0 0 004 1 0 : [LDUB]
0008551 : f80eee38 808ee01f 0000 1 0 0 004 1 0 : [ANDcc]
0008552 : f80eee30 10800015 0000 1 0 0 004 1 0 : [BA]
0008553 : f80eee34 ba102000 0000 1 0 0 004 1 0 : [OR]
0008554 : f80eee84 80a77fff 0000 1 0 0 004 1 0 : [SUBcc]
0008555 : f80eee88 2280000d 0000 1 0 0 004 1 0 : [BE]
0008556 : f80eee8c d6070000 0000 1 0 0 004 1 0 : [LD]
0008557 : f80eeebc 193c0000 0000 1 0 0 004 1 0 : [SETHI]
0008558 : f80eee90 153e05aa 0000 1 0 0 004 1 0 : [SETHI]
0008559 : f80eee94 9412a02c 0000 1 0 0 004 1 0 : [OR]
0008560 : f80eee98 bb2f6002 0000 1 0 0 004 1 0 : [SLL]
0008561 : f80eee9c c207400a 0000 1 0 0 004 1 0 : [LD]
0008562 : f80eeea0 9fc04000 0000 1 0 0 004 1 0 : [JMPL]
0008563 : f80eeea4 90100018 0000 1 0 0 004 1 0 : [OR]
0008564 : f816a82c f80f136c 0007 1 0 1 004 1 0 : [READ]/4
0008565 : f80eeea8 ba100008 0007 1 0 0 004 1 0 : [OR]
0008566 : f80f136c 9de3bfa0 0007 1 0 0 004 1 0 : [SAVE]
0008567 : f80f1370 f4062088 0013 1 0 0 004 1 0 : [LD]
0008568 : f80f1374 f606208c 0006 1 0 0 004 1 0 : [LD]
0008569 : f80f1378 fa56209e 0000 1 0 0 004 1 0 : [LSHW]
0008570 : fd801ec0 ff006000 0000 1 0 1 004 1 0 : [READ]/4
0008571 : f80f137c 900620a4 0000 1 0 0 004 1 0 : [ADD]
0008572 : fd801ec4 ff005000 0000 1 0 1 004 1 0 : [READ]/4
0008573 : f80f1380 bb2f6002 0007 1 0 0 004 1 0 : [SLL]
0008574 : fd801ed6 e60e203a 0006 1 0 1 002 1 0 : [READ]/2
0008575 : f80f138c 00000000 0003 1 0 1 002 1 0 : [READ]/2
```

APPENDIX B.

MODIFICATION LOG FILE CREATED BY BATE

```
Ref #069833 f8116fb0 da1f4000 0007 1 0 0 004 1 0  Fri Apr  5 16:15:40 1996
            f8116fb0 da0f4000 0007 1 0 0 004 1 0  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #146501 f8105040 e40e2060 0000 1 0 0 004 1 0  Sat Apr  6 16:22:01 1996
            f8105040 e41e2060 0000 1 0 0 004 1 0  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #146725 f8105064 e07f75e8 0000 1 0 0 004 1 0  Sat Apr  6 16:27:12 1996
            f8105064 e03f75e8 0000 1 0 0 004 1 0  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #146856 f810500c e00e20b8 0000 1 0 0 004 1 0  Sat Apr  6 16:29:34 1996
            f810500c e01e20b8 0000 1 0 0 004 1 0  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #146961 f8105098 e47f7510 0000 1 0 0 004 1 0  Sat Apr  6 16:31:48 1996
            f8105098 e43f7510 0000 1 0 0 004 1 0  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #244937 f8114564 e00e6001 0000 1 0 0 004 1 0  Sat Apr  6 17:30:17 1996
            f8114564 e0066001 0000 1 0 0 004 1 0  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #282817 f81052ec c20a0009 0007 1 0 0 004 1 0  Sun Apr  7 01:08:44 1996
            f81052c8 c20a0009 0007 1 0 0 004 1 0  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #282818 f81052f0 f227a048 0006 1 0 0 004 1 0  Sun Apr  7 01:10:31 1996
            f81052f0 f227a048 0006 1 0 0 004 1 1  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #282818 f81052f0 f227a048 0006 1 0 0 004 1 1  Sun Apr  7 01:12:59 1996
            00000000 f227a048 0006 1 0 0 004 1 1  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #282819 f81052cc c22a4000 0007 1 0 0 004 1 0  Sun Apr  7 01:18:09 1996
            f81052cc c22a4000 0014 1 0 0 004 1 0  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #303792 f81052d0 c43de070 0006 1 0 0 004 1 0  Sun Apr  7 01:47:47 1996
            f81052d0 c43de070 0006 1 0 0 004 1 1  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #303792 f81052d0 c43de070 0006 1 0 0 004 1 1  Sun Apr  7 01:48:03 1996
            00000000 c43de070 0006 1 0 0 004 1 1  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #303793 f81052d0 80900001 0007 1 0 0 004 1 0  Sun Apr  7 01:48:28 1996
            f81052d0 80900001 0014 1 0 0 004 1 0  [input/Sken1.00000]
-------------------------------------------------------------------------
Ref #375026 f810e900 e02bc000 0012 1 0 0 004 1 0  Mon Apr  8 04:28:45 1996
            f810e900 01000000 0012 1 0 0 004 1 0  [input/Sken1.00000]
-------------------------------------------------------------------------
```

APPENDIX C.

AN EXAMPLE CAPSIM LOG FILE

```
+-------------------------------------------------------------------+
| CaPSim Log File                                F. Nadir ALTMISDORT |
| Tue Sep  3 07:15:55 1996                                           |
+-------------------------------------------------------------------+

Starting configuration ---------------------------------------------

CPU             Reading Configuration File ...                : [OK]
CPU             Checking Syntax ...                           : [OK]
CPU             Setting Simulation Parameters ...             : [OK]
CPU             Checking Memory Hierarchy ...                 : [OK]
CPU             Checking Input/Output Paths ...               : [OK]
CPU             Starting Self-Test ...                        : [OK]

Initializing simulation module CacheL1                        : [ 1]
CacheL1         Cache Size                                    : [OK]
CacheL1         Block Size                                    : [OK]
CacheL1         SubBlock Size                                 : [OK]
CacheL1         Fetch Size                                    : [OK]
CacheL1         Transfer Size                                 : [OK]
CacheL1         Associativity                                 : [OK]
CacheL1         Replacement Policy                            : [OK]
CacheL1         Write Policy                                  : [OK]
CacheL1         Write Miss Policy                             : [OK]
CacheL1         Wrapping Fetch Policy                         : [OK]
CacheL1         Access Time                                   : [OK]
CacheL1         Read Hit Time                                 : [OK]
CacheL1         Read Miss Time                                : [OK]
CacheL1         Write Hit Time                                : [OK]
CacheL1         Write Miss Time                               : [OK]
CacheL1         Read Forward                                  : [OK]
CacheL1         Enable Block Buffer                           : [OK]
CacheL1         Search Block Buffer                           : [OK]
CacheL1         Block Buffer Transfer Time                    : [OK]
CacheL1         Starting Self-Test ...                        : [OK]

Initializing simulation module PRC                            : [ 2]
PRC             Prediction Algorithm                          : [OK]
PRC             PRC size                                      : [OK]
PRC             Block Size                                    : [OK]
PRC             Associativity                                 : [OK]
PRC             Replacement Policy                            : [OK]
PRC             Write Policy                                  : [OK]
PRC             Access Time                                   : [OK]
PRC             Read Hit Time                                 : [OK]
PRC             Read Miss Time                                : [OK]
PRC             Block Buffer Transfer Time                    : [OK]
PRC             Bypass Write Allocates                        : [OK]
PRC             Maximum read slips in buffer                  : [OK]
PRC             Minimum read size  in buffer                  : [OK]
PRC             Starting Self-Test ...                        : [OK]
```

```
Initializing simulation module Buffer1                          : [ 3]
Buffer1         Read  Buffer Size                               : [OK]
Buffer1         Write Buffer Size                               : [OK]
Buffer1         Write Buffer Block Size                         : [OK]
Buffer1         Enforce Priorities                              : [OK]
Buffer1         Remove Duplicates                               : [OK]
Buffer1         Starting Self-Test ...                          : [OK]

Initializing simulation module MainMemory                       : [ 4]
MainMemory      Access   Time                                   : [OK]
MainMemory      Transfer Time                                   : [OK]
MainMemory      Transfer Size                                   : [OK]
MainMemory      Starting Self-Test ...                          : [OK]

Finalizing simulation modules ...                               :
CPU             Finalize ...                                    : [OK]
CacheL1         Finalize ...                                    : [OK]
PRC             Finalize ...                                    : [OK]
Buffer1         Finalize ...                                    : [OK]
MainMemory      Finalize ...                                    : [OK]

CaPSim configuration completed successfully @ Tue Sep  3 07:15:55 1996

Starting simulation ------------------------------------------------

Opening file    input/Sken1.00000                               : [OK]
Opening file    input/Sken1.00001                               : [OK]
Opening file    input/Sken1.00002                               : [OK]
Opening file    input/Sken1.00003                               : [OK]
Opening file    input/Sken1.00004                               : [OK]
Opening file    input/Sken1.00005                               : [OK]
Opening file    input/Sken1.00006                               : [OK]
Opening file    input/Sken1.00007                               : [OK]
Opening file    input/Sken1.00008                               : [OK]
Opening file    input/Sken1.00009                               : [OK]
Opening file    input/Sken1.00010                               : [OK]

The simulation is completed successfully    @ Wed Sep  4 00:10:10 1996

Dumping simulation modules ...                                  :
CPU             Dumping dPRC_128k/CPU_dump.00099                : [OK]
CacheL1         Dumping dPRC_128k/CacheL1_dump.00099            : [OK]
PRC             Dumping dPRC_128k/PRC_dump.00099                : [OK]
Buffer1         Dumping dPRC_128k/Buffer1_dump.00099            : [OK]
MainMemory      Dumping dPRC_128k/MainMemory_dump.00099         : [OK]

Closing Log File ......................... @ Wed Sep  4 00:10:11 1996
```

APPENDIX D.

AN EXAMPLE OUTPUT FILE FOR THE CPU CLASS

```
+-------------------------------------------------------------------------+
| Module Title    : CPU                                                   |
| Module ID       : 0                                                     |
| Configuration   : iPRC_32k                    Fri Sep  6 01:33:29 1996 |
+-------------------------------------------------------------------------+

System Clock : 0073749152 ---------------------------------------------

Operating Parameters --------------------------------------------------

Number of Simulation Modules        : 5
Word Size                           : 4
Trace Type                          : PRC Trace
Trace Filename                      : output/skenPRC.00099
Start File Number                   : 0
Stop  File Number                   : 99
Maximum Trace Buffer Size           : 2000
Current Trace Buffer Index          : 1893
Last Entry in Trace Buffer          : 1893

Simulation Set --------------------------------------------------------

+--------------------+---+
| CPU                | 0 |
+--------------------+---+
| CacheL1            | 1 |
+--------------------+---+
| PRC                | 2 |
+--------------------+---+
| Buffer1            | 3 |
+--------------------+---+
| MainMemory         | 4 |
+--------------------+---+

Event Queue Contents --------------------------------------------------

+-----------------------+
| CaPSim    Event Queue |
| Size: 01 @ 0073749152 |
+-----------------------+
| Module # | Event Time |
+----------+------------+
|    04    | 0073749152 |
+----------+------------+
```

```
Number of Canceled Events : 37474

Module States ------------------------------------------------------------

CPU            State @0073749152 : WriteStall
CacheL1        State @0073749152 : Idle       Block Buffer : Idle
PRC            State @0073749152 : Idle       Block Buffer : Idle
Buffer1        State @0073749152 : W-Access
MainMemory     State @0073749152 : Access

Statistics ---------------------------------------------------------------

Total Number of Requests           : 7121893
Total Number of Read  Requests     : 4900537
Total Number of Write Requests     : 2221356
Total Read  Stall Cycles           : 6738930
Total Write Stall Cycles           : 2221356
Average Read  Access Time          : 1.37514114
Average Write Access Time          : 1.00000000

END OF FILE [iPRC_32k/CPU_dump.00099] ------------------------------------
```

APPENDIX E.

AN EXAMPLE OUTPUT FILE FOR THE CACHE CLASS

```
+-------------------------------------------------------------------------+
| Module Title    : CacheL1                                               |
| Module ID       : 1                                                     |
| Configuration   : iPRC_32k                    Fri Sep  6 01:33:29 1996 |
+-------------------------------------------------------------------------+

System Clock : 0073749152 ------------------------------------------------

Operating Parameters -----------------------------------------------------

Cache Size                          : 65536
Block Size                          : 16
Sub-Block Size                      : 4
Fetch Size                          : 16
Transfer Size                       : 4
Associativity                       : 1 (Direct-Mapped)
Number of Sets                      : 4096
Total Number of Blocks              : 4096
Number of Sub-Blocks                : 4
Replacement Policy                  : LRU
Write Policy                        : Write Through
Write Miss Policy                   : Write Around
Wrapping Fetch Policy               : Wrap Up
Start Policy                        : Cold Start
Read Forward                        : Yes
Enable Block Buffer                 : Yes
Search Block Buffer                 : Yes
Read  Access Time                   : 1
Write Access Time                   : 1
Read  Hit    Time                   : 0
Read  Miss   Time                   : 0
Write Hit    Time                   : 0
Write Miss   Time                   : 0
Block Buffer Transfer Time          : 1

Address Decoder ----------------------------------------------------------

+----------------+------------+--+--+ +-----------------+
|3322222222221111|111111000000|00|00| |t : tag  bits = 16|
|1098765432109876|543210987654|32|10| |s : set  bits = 12|
+----------------+------------+--+--+ |w : word bits = 02|
|tttttttttttttttt|ssssssssssss|ww|bb| |b : byte bits = 02|
+----------------+------------+--+--+ +-----------------+
```

```
Block Address     Mask : fffffff0 hex
Sub-block Address Mask : fffffffc hex
Word Address      Mask : fffffffc hex
Set Number        Mask : 0000fff0 hex
Sub-block Number  Mask : 0000000c hex
Word Number       Mask : 0000000c hex
Word Byte Number  Mask : 00000003 hex
Block Byte Number Mask : 0000000f hex

Statistics ----------------------------------------------------------------

Total Number Of Read  Requests      : 4900537
Total Number Of Write Requests      : 2221356
Number Of Read  Requests            : 4900537
Number Of Write Requests            : 2221356
Number Of Read  Cancels             : 0
Number Of Write Cancels             : 0
Number Of Read  Hits                : 4341931
Number Of Write Hits                : 1419684
Number Of Dirty Read  Misses        : 0
Number Of Dirty Write Misses        : 0

Global Read  Hit  Ratio             : 0.88601124
Global Read  Miss Ratio             : 0.11398876

Global Write Hit  Ratio             : 0.63910693
Global Write Miss Ratio             : 0.36089307

Local  Read  Hit  Ratio             : 0.88601124
Local  Read  Miss Ratio             : 0.11398876

Local  Write Hit  Ratio             : 0.63910693
Local  Write Miss Ratio             : 0.36089307

Dirty Read  Miss Ratio              : 0.00000000
Dirty Write Miss Ratio              : 0.00000000
Dirty Read  Miss Percentage         : 0.00000000 %
Dirty Write Miss Percentage         : 0.00000000 %

Read Miss Cycles                    : 2312214
Read Miss Penalty                   : 4.13925743

Block Buffer Read  Hits             : 20750
Block Buffer Write Hits             : 0

Block Buffer Read  Hit Ratio        : 0.00423423
Block Buffer Write Hit Ratio        : 0.00000000

END OF FILE [iPRC_32k/CacheL1_dump.00099] ---------------------------------
```

APPENDIX F.

AN EXAMPLE OUTPUT FILE FOR THE PRC CLASS

```
+-------------------------------------------------------------------------+
| Module Title    : PRC                                                   |
| Module ID       : 2                                                     |
| Configuration   : iPRC_32k                    Fri Sep  6 01:33:30 1996 |
+-------------------------------------------------------------------------+

System Clock : 0073749152 ---------------------------------------------

Operating Parameters --------------------------------------------------

PRC Algorithm                     : Instruction Address Displacement
PRC Size                          : 32768
Block Size                        : 16
Sub-Block Size                    : 16
Fetch Size                        : 16
Transfer Size                     : 16
Associativity                     : 2048 (Fully-Associative)
Number of Sets                    : 1
Total Number of Blocks            : 2048
Number of Sub-Blocks              : 1
Replacement Policy                : LRU
Write Policy                      : Write Through
Write Miss Policy                 : Write Around
Bypass Write Allocates            : Yes
Read  Access Time                 : 0
Write Access Time                 : 0
Read  Hit    Time                 : 1
Read  Miss   Time                 : 1
Write Hit    Time                 : 0
Write Miss   Time                 : 0
Block Buffer Transfer Time        : 1

Address Decoder -------------------------------------------------------

INSTRUCTION ADDRESS DECODER :

+------------------------------+--+ +-----------------+
|33222222222211111111100000000|00| |t : tag  bits = 30|
|10987654321098765432109876543|10| |s : set  bits = 00|
+------------------------------+--+ +-----------------+
|ttttttttttttttttttttttttttttt|00|
+------------------------------+--+

Instruction Tag   Mask : fffffffc hex
```

```
Instruction Set   Mask : 00000000 hex

DATA ADDRESS DECODER :

+----------------------------+--+--+ +-----------------+
|33222222222211111111110000000|00|00| |t : tag  bits = 1c|
|10987654321098765432109876543|32|10| |s : set  bits = 00|
+----------------------------+--+--+ |w : word bits = 02|
|ttttttttttttttttttttttttttttt|ww|bb| |b : byte bits = 02|
+----------------------------+--+--+ +-----------------+

Block Address     Mask : fffffff0 hex
Sub-block Address Mask : fffffff0 hex
Word Address      Mask : fffffffc hex
Set Number        Mask : 00000000 hex
Sub-block Number  Mask : 00000000 hex
Word Number       Mask : 0000000c hex
Word Byte Number  Mask : 00000003 hex
Block Byte Number Mask : 0000000f hex

Statistics ----------------------------------------------------------------

Total Number Of Read  Requests      : 4900537
Total Number Of Write Requests      : 2221356
Number Of Read  Requests            : 537856
Number Of Write Requests            : 2221356
Number Of Read  Cancels             : 2040
Number Of Write Cancels             : 0
Number Of Read  Hits                : 253288
Number Of Write Hits                : 739542
Number Of Transfer Stalls           : 1857

Total   Hits                        : 253084
Partial Hits                        : 204
Total   Misses                      : 3667
Partial Misses                      : 280901
Maximum Write Hits                  : 21

Number Of Prefetch Requests         : 463685
Number Of Invalid Predictions       : 68586
Wrap-Around From Left               : 507
Wrap-Around From Right              : 64
Prediction in the Same Block        : 68015
Maximum Pending Prefetches          : 10199

Global Read  Hit  Ratio             : 0.05168577
Global Read  Miss Ratio             : 0.94831425

Global Write Hit  Ratio             : 0.33292368
Global Write Miss Ratio             : 0.66707635
```

```
Local  Read  Hit  Ratio            : 0.47092158
Local  Read  Miss Ratio            : 0.52907842

Local  Write Hit  Ratio            : 0.33292368
Local  Write Miss Ratio            : 0.66707635

Block Buffer Read  Hits            : 8
Block Buffer Write Hits            : 7

Block Buffer Read  Hit Ratio       : 0.00001487
Block Buffer Write Hit Ratio       : 0.00000315

END OF FILE [iPRC_32k/PRC_dump.00099] -----------------------------------
```

APPENDIX G.

AN EXAMPLE OUTPUT FILE FOR THE BUFFER MODULE CLASS

```
+-------------------------------------------------------------------------+
| Module Title    : Buffer1                                               |
| Module ID       : 3                                                     |
| Configuration   : iPRC_32k                    Fri Sep  6 01:33:30 1996 |
+-------------------------------------------------------------------------+

System Clock : 0073749152 ---------------------------------------------

Operating Parameters --------------------------------------------------

Read  Buffer Size                 : 8
Write Buffer Size                 : 4
Write Buffer Block Size           : 16
Enforce Priorities                : Yes
Remove Read  Duplicates           : Yes
Remove Write Duplicates           : Yes
Search Read  Buffer               : Yes
Search Write Buffer               : Yes

Read Buffer Contents --------------------------------------------------

+---------------------------------+
| READ  BUFFER [EMPTY] : 0/8      |
| Access In Progress   : No       |
| # Pushes Attempted   : 748237   |
| # Pushes Granted     : 748043   |
| # Pushes Rejected    : 194      |
+---------------------------------+

Write Buffer Contents -------------------------------------------------

+---------------------------------+
| WRITE BUFFER         : 1/4      |
| Access In Progress   : Yes      |
| # Pushes Attempted   : 1890861  |
| # Pushes Granted     : 1890861  |
| # Pushes Rejected    : 0        |
+---------------------------------+
| Source Module ID     : 1        |
| Transaction Type     : Write    |
| Transaction Size     : 4        |
| Data Address         : f8191ef4 |
| Instruction Address  : f810b5ec |
| Transaction Priority : 12       |
```

```
| Minimum Size          : Disabled |
| Drop Counter          : Disabled |
+----------------------------------+

Statistics ------------------------------------------------------------------

Total Number Of Read  Requests      : 4900537
Total Number Of Write Requests      : 2221356
Number Of Read  Requests            : 748245
Number Of Write Requests            : 2221356

READ  BUFFER :
Number of Requests Slipped          : 35628
Number of Requests Dropped          : 1846
Total Number of Matches             : 7796
Number of Matches (Low-High)        : 0
Number of Matches (High-Low)        : 7796
Instruction Addres Matches          : 8
Victim Block Matches                : 0
Total   Write Hits                  : 0
Partial Write Hits                  : 0

WRITE BUFFER :
Number of Inclusive Merges          : 0
Number of Adjacent  Merges          : 320667
Total Number of Matches             : 0
Number of Matches (Low-High)        : 0
Number of Matches (High-Low)        : 0
Total   Read Hits                   : 0
Partial Read Hits                   : 0

END OF FILE [iPRC_32k/Buffer1_dump.00099] -------------------------------
```

APPENDIX H.

AN EXAMPLE OUTPUT FILE FOR THE MAIN MEMORY CLASS

```
+-------------------------------------------------------------------------+
| Module Title    : MainMemory                                            |
| Module ID       : 4                                                     |
| Configuration   : iPRC_32k                    Fri Sep  6 01:33:30 1996 |
+-------------------------------------------------------------------------+

System Clock : 0073749152 ---------------------------------------------

Operating Parameters --------------------------------------------------

Memory Access   Time                : 5
Memory Transfer Time                : 1

Statistics ------------------------------------------------------------

Number Of Read  Requests            : 775525
Number Of Write Requests            : 1568664
Number Of Read  Cancels             : 37474
Number Of Write Cancels             : 0

Total Number Of Cycles              : 13971821
Number Of Idle  Cycles              : 59777331
Number Of Read  Cycles              : 5952922    [42.61 %]
Number Of Write Cycles              : 8018899    [57.39 %]

Total Memory Utilization            : 0.18945059
Memory Read  Utilization            : 0.08071852
Memory Write Utilization            : 0.10873208

Average Read  Service Time          : 7.67598963
Average Write Service Time          : 5.11192894
Global  Read  Service Time          : 1.21474886
Global  Write Service Time          : 3.60991168

END OF FILE [iPRC_32k/MainMemory_dump.00099] ---------------------------
```

LIST OF REFERENCES

1. Patterson, D. A. and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

2. Przybylski, S. A., *Cache and Memory Hierarchy Design: A Performance Directed Approach*, Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

3. Smith, A. J., "Line (Block) Size Choice for the CPU Cache Memories," *IEEE Transactions on Computers*, vol. C-36, no. 9, September 1987, pp. 1063-1075.

4. Patterson, D. A. and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufman Publishers, Inc., San Francisco, CA, 1994.

5. Handy, J., *The Cache Memory Book*, Academic Press Inc., San Diego, CA, 1993.

6. Fouts, D. J. and A. B. Billingsley, "Predictive Read Caches: An Alternative to On-Chip Second-Level Cache Memories," *Journal of Microelectronic Systems Integration*, vol. 2, no. 2, 1994.

7. Billingsley, A. B., "An Investigation of Memory Latency Reduction Using an Address Prediction Buffer," Master's Thesis, U.S. Naval Postgraduate School, Monterey, CA, December 1992.

8. Miller, R. W., "Simulation and Analysis of Predictive Read Cache Performance," Master's Thesis, U.S. Naval Postgraduate School, Monterey, CA, June 1995.

9. Grimsrud, K., J. Archibald, M. Ripley, K. Flanagan, and B. Nelson, "BACH: A Hardware Monitor for Tracing Microprocessor-Based Systems," *Microprocessors and Microsystems*, vol. 17, no. 8, October 1993.

10. Weste, Neil H. E. and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley Publishing Company, Burlington, MA, October 1994.

11. Agarwal, A., J. Hennessy, and M. Horowitz, "Cache Performance of Operating System and Multiprogramming Workloads," *ACM Transactions on Computer Systems*, vol. 6, no. 4, pp. 393-431, November 1988.

12. Stone, S. S., *High-Performance Computer Architecture*, Addison-Wesley Publishing Company, Menlo Park, CA, October 1987.

13. Smith, A. J., "Cache Evaluation and the Impact of Workload Choice," *Proceedings of the Twelfth International Symposium on Computer Architecture*, pp. 64-73, 1985.

14. Flanagan, J. K., B. E. Nelson, and G. Thompson, "The Inaccuracy of Trace-Driven Simulation Using Incomplete Multiprogramming Trace Data," *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems MASCOTS*, pp. 37-43, 1996.

15. Catanzaro, B., *Multiprocessor System Architectures*, Sun Microsystems Inc., Mountain View, CA, 1994.

16. *The SPARC Architecture Manual Version 8*, SPARC International Inc., Menlo Park, CA, 1992.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2.	Dudley Knox Library Naval Postgraduate School 411 Dyer Rd. Monterey, California 93943-5121	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
4.	Professor Herschel H. Loomis, Jr., Code EC/Lm Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
5.	Professor Douglas J. Fouts, Code EC/Fs Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	2
6.	Professor Frederick W. Terman, Code EC/Tz Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	2
7.	Professor Donald Brutzman, Code CC/Bz Naval Postgraduate School Monterey, California 93943-5122	1
8.	Deniz Kuvvetleri Komutanligi Personel Daire Bsk.ligi Bakanliklar, Ankara Turkey	1

9. Kara Harp Okulu Komutanligi 1
Kutuphane
Bakanliklar, Ankara
Turkey

10. Deniz Harp Okulu Komutanligi 1
Kutuphane
Tuzla, Istanbul
Turkey

11. Hava Harp Okulu Komutanligi 1
Kutuphane
Yesilyurt, Istanbul
Turkey

12. Golcuk Tersanesi Komutanligi 1
Golcuk, Kocaeli
Turkey

13. Taskizak Tersanesi Komutanligi 1
Kasimpasa, Istanbul
Turkey

14. Orta Dogu Teknik Universitesi 1
Universite Kutuphanesi
Balgat, Ankara
Turkey

15. Bogazici Universitesi 1
Universite Kutuphanesi
Bebek, Istanbul
Turkey

16. Istanbul Teknik Universitesi 1
Universite Kutuphanesi
Macka, Istanbul
Turkey

17. Dokuz Eylul Universitesi
Universite Kutuphanesi
Izmir
Turkey — 1

18. Ltjg F. Nadir Altmisdort
Acibadem Cad. Genc Sok.
Ozsarac Apt. 16/3
81020 Kadikoy Istanbul
Turkey — 3